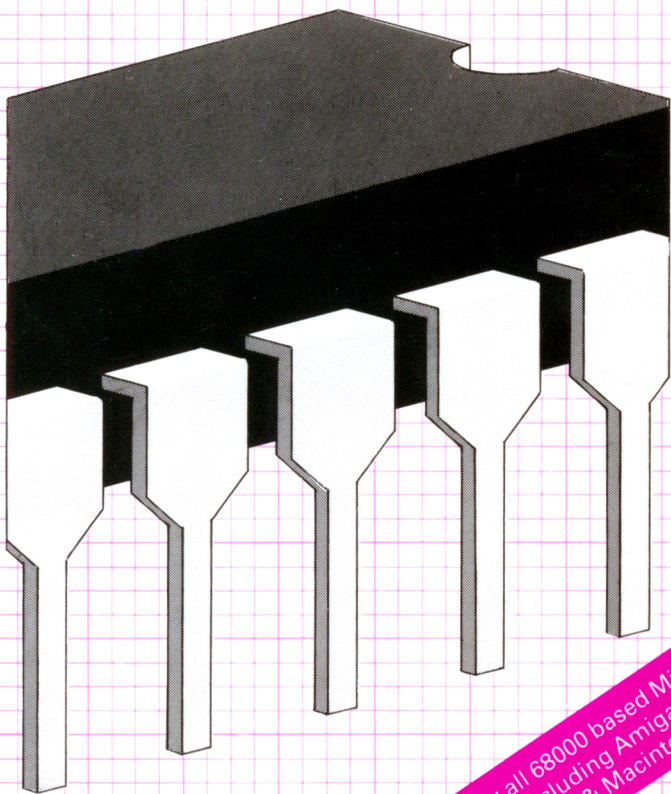


An Introduction to 68000 Assembly Language

R.A. & J.W. PENFOLD



For all 68000 based Micros
including Amigas,
Atari STs & Macintoshes etc

**AN INTRODUCTION TO
68000 ASSEMBLY LANGUAGE**

OTHER BOOKS OF INTEREST

- BP112 A Z80 Workshop Manual
- BP147 An Introduction to 6502 Machine Code
- BP150 An Introduction to Programming the Sinclair QL
- BP152 An Introduction to Z80 Machine Code
- BP156 An Introduction to QL Machine Code

**AN INTRODUCTION TO
68000 ASSEMBLY LANGUAGE**

by

R. A. & J. W. PENFOLD

**BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND**

PLEASE NOTE

Although every care has been taken with the production of this book to ensure that any projects, designs, modifications and/or programs etc. contained herewith, operate in a correct and safe manner and also that any components specified are normally available in Great Britain, the Publishers do not accept responsibility in any way for the failure, including fault in design, of any project, design, modification or program to work correctly or to cause damage to any other equipment that it may be connected to or used in conjunction with, or in respect of any other damage or injury that may be so caused, nor do the Publishers accept responsibility in any way for the failure to obtain specified components.

Notice is also given that if equipment that is still under warranty is modified in any way or used or connected with home-built equipment then that warranty may be void.

© 1986 BERNARD BABANI (publishing) LTD

First Published — August 1986

British Library Cataloguing in Publication Data:

Penfold, R.A.

An introduction to 68000 assembly language.

1. Assembler language (Computer program language)

2. 68000 (Computer program language)

I. Title II. Penfold, J.W.

005.13;6 QA76.73.S6/

ISBN 0 85934 158 5

PREFACE

Home computers are equipped with built-in software that enables them to be easily programmed to do quite complex tasks. The price that is paid for this programming ease is a relatively slow running speed, far slower than the speed at which the computer is really capable of running. Assembly language programming entails direct programming of the computer without using a built-in high level language such as BASIC. This gives a vast increase in running speed, but is something that can only be undertaken by someone who has a reasonable understanding of the microprocessor and some of the other hardware in the computer.

Assembly language programming is not as difficult as one might think, and once a few fundamental concepts have been grasped it is actually quite straightforward (although admittedly never as quick and easy as using a high level language). This book takes the reader through the basics of microprocessors and assembly language programming, and no previous knowledge of either of these is assumed. The microprocessor dealt with is the 68000 (and its derivatives such as the 68008), which is widely acknowledged as one of the most powerful types currently available (possibly the most powerful of all). This has led to it being used as the basis of an increasing number of home and business computers. A point well worth making is that the advanced specification of the 68000 does not mean that it is correspondingly difficult to program. In fact it is in many ways more easy to program than the popular 8 bit microprocessors, and is not one purely the domain of advanced programmers.

R.A. Penfold

CONTENTS

	Page
Chapter 1: THE MICROPROCESSOR	1
Assembly Language	2
The Processor	5
Binary	11
Signed Binary	14
Twos Complement	16
Overflow Flag	17
Binary Coded Decimal	18
Hexadecimal	20
System Operation	22
The Stack	26
Flags	28
Chapter 2: ADDRESSING MODES	30
1. Implicit	30
2. Register Addressing	30
3. Immediate Addressing	31
4. Absolute Short Addressing	31
5. Absolute Long Addressing	32
Address Register Indirect Addressing	32
6. Address Register Indirect with Displacement	33
7. Address Register Indirect with Index and Displacement	33
8. Address Register Indirect with Postincrement	34
9. Address Register Indirect with Predecrement	34
10. Program Counter Relative with Displacement	35
11. Program Counter Relative with Index and Displacement	35
Chapter 3: THE INSTRUCTION SET	36
Chapter 4: ASSEMBLERS	84
Assembly Language Formatting	85
Labels	86
Assembler Directives	88

	Page
Chapter 5: SIMPLE PROGRAMS	91
Addition	91
Subtraction	93
Multiplication	94
Division	95
Loops	97
Clearing a Section of Memory	98

Chapter 1

THE MICROPROCESSOR

All home and personal computers are equipped to operate using a high level computer language such as BASIC or FORTH, and these languages are designed to make program design as quick and easy as possible. With most high level languages the programmer uses words that are virtually plain English, and the computer's built-in software then converts these into machine code routines that the microprocessor at the heart of the computer can interpret and act upon. Writing programs direct in machine code is, on the face of it, rather pointless, as it is somewhat harder and a considerably slower process than using BASIC or another high level language to achieve the same ends.

The advantage of machine code programs is the speed with which they run. The speed of a machine code program is, in fact, only limited by the operating speed of the computer's microprocessor, and a computer can perform no faster than when it is running a machine code program. High level languages such as BASIC are inevitably much slower due to the way in which each instruction has to first be interpreted (converted into machine code) before it can be executed. In other words, the program is stored in memory in its BASIC form, and it is only when the program is run that each instruction is converted into machine code and executed. The program is effectively brought to a halt during the interpreting process, which accounts for more time than the running of the interpreted machine code. The difference in speed is probably much greater than most people realise, and machine code is typically something approaching one thousand times faster than an equivalent BASIC program. Action games written in any form of BASIC are usually a little sluggish due to this lack of operating speed, especially when a lot starts to happen at once, but a machine code equivalent normally appears to operate instantly no matter how much simultaneous action takes place. With some scientific and business programs BASIC is too slow to be of any use

at all, and the use of machine code is mandatory. However, the speed of machine code is its only advantage, and apart (perhaps) for the fun of it, there is no point in using machine code where a program written in a high level language would be fast enough.

There are alternatives to machine code and high level interpreted languages such as BASIC, and we will consider these briefly before moving on to a description of the microprocessor itself. Some high level languages are compiled rather than interpreted. The difference is that with a compiled language the interpreting process is carried out before the program is run. The program may then run using the compiled machine code, or using a sort of pseudo machine code which requires a minimal amount of interpreting. In either case programs should run at high speed, and should be far easier to write than equivalent machine code programs. A compiled language may seem like the ideal solution (and many people would argue that it is), but languages of this type are generally much more difficult to use than interpreted languages when writing and debugging programs, and languages such as BASIC are probably much better for beginners to programming. A mixture of BASIC and machine code (with the latter only being used where high operating speed is essential) can therefore be a more practical solution in many cases.

Incidentally, you may come across the terms "source code" and "object code" occasionally. The former is the program in its high level language form, and the latter is the machine code or pseudo machine code produced after interpretation or compilation.

Assembly Language

The terms "machine code" and "assembly language" seem to cause a certain amount of confusion, and there seems to be a general belief that they are different terms for the same thing. In fact they are very similar, but there is an important difference. When machine code programming, the instructions for the microprocessor are in the form of numbers which are usually from 0 to 255, but in the case of the advanced 68000

series of microprocessors the numbers are from 0 to 65535. In fact some instructions require more than one code number. This is not a very convenient way of doing things, and it inevitably involves almost constantly looking up instructions to find their code numbers.

Assembly language uses a program in the computer to take three or four letter codes and convert these into the corresponding machine code instruction numbers. Most “assemblers” also provide the programmer with some further assistance, but not much when compared to a high level language such as BASIC. The main function of the assembler is simply to take the three or four letter “mnemonics”, convert them to the appropriate numbers, and store these in the required area of the computer’s memory. An assembler is really the most basic of compilers, but as far as the programmer is concerned there is no real difference between assembly language and machine code apart from programming ease, and if you can program in one you can also program using the other.

Of course, the main advantage of using an assembler is that the mnemonics are chosen to closely relate to the instructions that they represent. For example, the “Return From Subroutine” instruction has “RTS” as its mnemonic, which is obviously much easier to remember than the machine code number of 20085. If you intend to do a lot of machine code programming an assembler could reasonably be considered essential, since using anything other than a few short machine code routines is generally rather awkward and inconvenient with a home or personal computer which was designed primarily for programming in a high level language. This is especially the case with a computer that has an advanced microprocessor such as one of the 68000 series, which makes assembly language programming in many ways a relatively simple task, but where machine code programming is a very slow and tedious task indeed. The facilities offered vary somewhat from one assembler to another, but most give at least some aid with debugging, although they are nothing like as sophisticated as the best BASIC languages in this respect. Some assembly language routines are included in the final section of this book so that you can try out a few programs,

and further information on assemblers and their use is included in this section. Your 68000 based computer must be loaded with an assembler in order to run any of these programs.

One final point to bear in mind is that a high level language such as BASIC varies only slightly from one computer to another, and once you have mastered BASIC it is usually not too difficult to write programs for any computer equipped with some form of this language. Problems can arise with the sound and graphics facilities which vary from one machine to another, giving inevitable variations in the sound and graphics commands, and some versions are more comprehensive than others. However, the language is fundamentally the same for all the computers that use it. Machine code programming is identical for any computers that use the 68000 microprocessor (or a derivative) as the central processor. Although there are again differences in the sound and graphics facilities available on various machines, these do not affect the instructions that are available to the programmer (although to produce the desired effect it might be necessary to use a different routine for each machine because of differences in the supporting hardware for the microprocessor). The situation is very different when dealing with a computer that uses a different microprocessor such as the 6502. Apart from the differences in the sound and graphics facilities, the microprocessor will have different machine code numbers for each instruction, and probably even different mnemonics. For instance, the 68000 Return From Subroutine instruction, as mentioned earlier, has RTS as its mnemonic, and 20085 is the instruction number. The equivalents for the popular Z80 microprocessor are RET and 169. Furthermore, the instruction sets of various microprocessors are substantially different, as are the registers they contain and the way in which they handle certain tasks. Obviously all microprocessors work on the same basic principle, but rewriting a machine code program to run on a different microprocessor is not usually just a matter of converting the mnemonics or code numbers, and changing from programming one type to programming an alternative device usually involves a fairly substantial amount of work. In practice this means that you

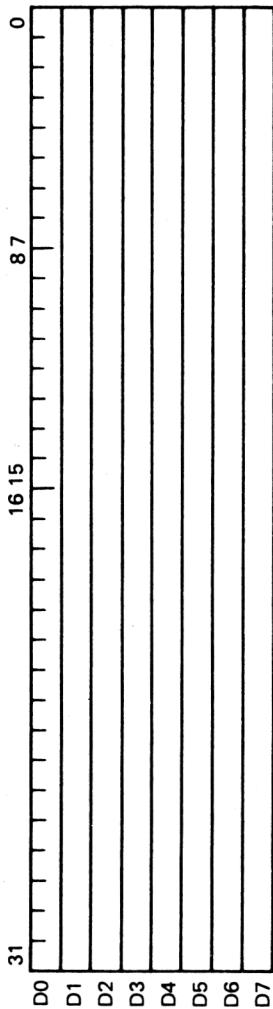
should be able to program any 68000 series based computer after conquering machine programming for these microprocessors, but might find it difficult to program other computers using machine code.

Some readers might be confused by references to the 68000 series of microprocessors, rather than just to the 68000. This is necessary because there are several versions of this microprocessor, and the Sinclair QL computer for example, uses the 68008 with its eight bit data bus instead of the standard sixteen bit type. As far as the assembly language programmer is concerned this is all of academic importance, and 68000 assembly language programs should run using any version of the processor. The 68008 has to deal with 16 bit words of memory as two 8 bit bytes, but the microprocessor takes care of this and it is something that is not apparent to the user.

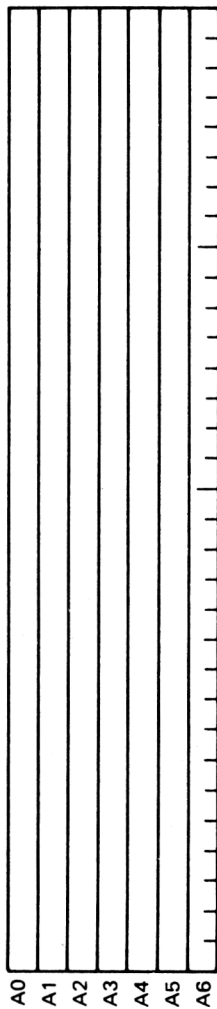
The Processor

Although a microprocessor is an extremely complex device, usually containing the equivalent of tens of thousands of components, as far as the programmer is concerned it can be regarded as a fairly simple set of electrical circuits known as "registers" which will perform certain functions if fed with the appropriate instruction numbers. The registers consist of one or more circuits known as flip/flops, and these can produce an output voltage that is either virtually zero, or one that is typically about 5 volts. From the software point of view the voltages are not important, and we can think in terms of "low" or logic 0 if the output of a flip/flop is near zero volts, and "high" or logic 1 if it is at about 5 volts. A circuit with an output that can represent just 0 or 1 may not seem to be very useful, and in isolation such a circuit is not of tremendous value, but as we shall see later, a number of flip/flops together can represent large numbers, and can be used to perform complex calculations etc.

The registers of the 68000 are shown in diagramatic form in Figure 1, and this diagram may be a little meaningless to you at this stage. In fact the register set of the 68000 is a little



Eight 32
bit data
registers



Seven
32 bit
address
registers

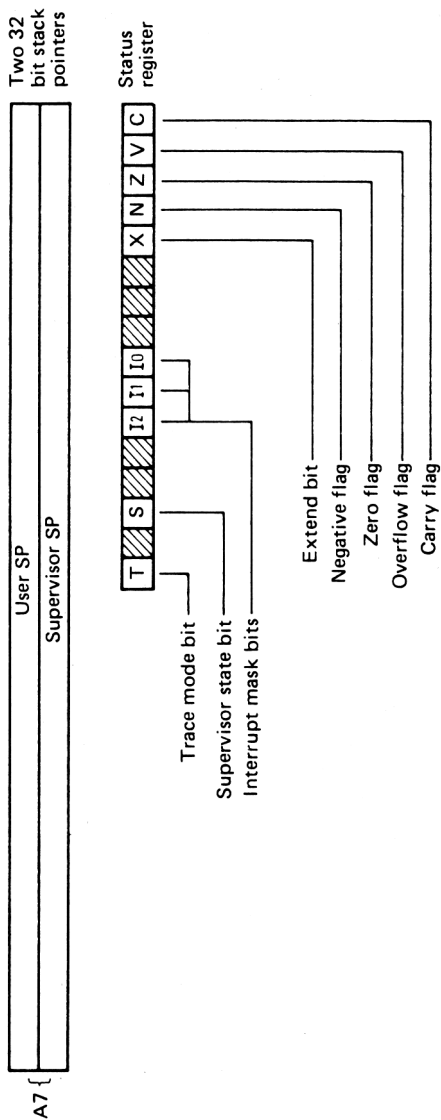


Fig. 1. The 68000 register set

unusual by conventional standards, and this diagram would probably be meaningless to someone who is only familiar with one of the popular 8 bit microprocessors such as the 6502 or the Z80. Most microprocessors have a register called the "accumulator", or in some cases there are two of these registers. The accumulator is an extremely important register since any manipulation of data (addition, subtraction, etc.) normally takes place in this register, or to be more accurate, the result of any data manipulation is placed in this register. The calculations are actually handled by the arithmetic logic unit (ALU), but this is something with which the programmer does not become directly involved. If you give the microprocessor certain instructions it carries them out and performs a given task. Exactly how it manages to do this is something that the programmer does not usually need to understand.

Looking at the register set there is an obvious omission in that there is no accumulator. In fact the 68000 is a highly advanced microprocessor which has the eight data registers instead. These can all operate as accumulators, and can also act as index registers or counters (something we will consider in more detail later).

The seven address registers can be used as index registers, or as straightforward address registers. They are not normally used to hold data. In order to understand the function of the various registers it is really necessary to understand, amongst other things, the basic make-up of a computer. Figure 2 shows in block diagram form the general arrangement used in a 68000 based computer. The memory is a bank of 16 bit registers which are used to store both program instructions and data. The number of registers in the memory block varies from one computer to another, but the 68000 can operate with a maximum of 16777216 (the cut down address bus of the 68008 permits 1048576 registers to be addressed, which still permits a massive amount of memory to be used). The address bus is 24 bits wide (20 bits on the 68008), and these bits are produced by the program counter (see Fig. 1). It is the program counter, via the address bus, that selects the particular memory register that is connected to the microprocessor. The data bus is used to transfer data between the microprocessor and the memory block. An important point

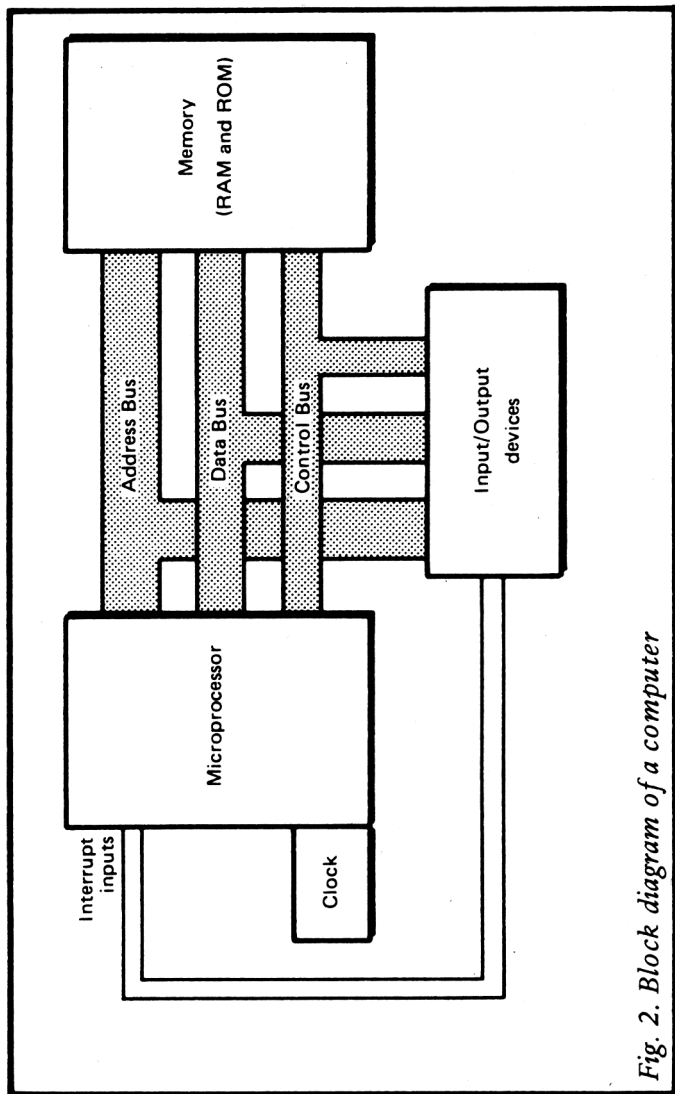


Fig. 2. Block diagram of a computer

to note here is that the data bus is bi-directional, and is used by the microprocessor to take data and instructions from memory, and to place data in memory. There are not separate input and output busses on a microprocessor — the data bus is used for both types of operation.

Most microprocessors have 8 bit registers, with some possibly arranged in pairs to effectively form 16 bit registers. Apart from the status register, all the 68000's registers are 32 bits long. This enables large numbers to be handled much more easily and rapidly than is possible using an 8 bit microprocessor, and is certainly a big advantage in many practical applications. Most of the microprocessors in the 68000 range have a 16 bit wide data bus, and therefore have to take in and send out 32 bit chunks of data in two sections to the 16 bit memory registers. The 68008 has only an 8 bit data bus, and therefore has to handle 32 bit chunks of data in four sections (and the memory registers are only 8 bit types). As mentioned earlier, as far as the programmer is concerned this is only of academic importance since the microprocessor automatically takes in or outputs four 8 bit pieces of data, and to a large extent there is no difference between 68000 and 68008 programs. The hardware is obviously different, but this is not of importance to the programmer. The 8 bit data bus of the 68008 does make it somewhat slower in operation than the devices in the 68000 series which have a 16 bit data bus, but this is unlikely to be of any significance in practice.

The 68000 does not have to deal with 32 bit blocks of data, and it can handle data in 8 or 16 bit chunks where this would be more convenient. A collection of 8 bits of data is normally termed a "byte", and this is the size in which most microprocessors handle data. Things are different in the case of the 68000 with its 16 and 32 bit capability, and a different terminology is needed. The system that seems to have become established, and the one we will use in this book, is to retain the word "byte" for 8 bits, and to augment this with "word" for 16 bits of data, and "long word" to cover 32 bits of data. The way in which numbers can be represented by a series of 1s and 0s is something we will consider shortly.

Returning to Figure 2, the control bus is used to make sure that all the elements of the system are operating in unison, and

that if (say) the microprocessor sends data to a particular register in memory, that register is ready to receive the data and is not trying to output data to the microprocessor. All the lines in the control bus operate automatically, are not directly controlled by the programmer, and are not something we need concern ourselves with here.

Binary

The 24 bit program counter can place 16777216 different output combinations onto the address bus, and it is this that limits the 68000 to 16777216 memory registers. Each memory register occupies an "address", which is merely a number from 0 to 16777216, and each of the output combinations of the program counter corresponds to one of these addresses. Therefore, by placing each bit of the program counter at the appropriate state, the microprocessor can read the contents of any memory register, or can write data to that register, depending on the type of instruction it is executing. In order to undertake machine code or assembly language programming it is essential to understand the way in which the bits of the address bus (and the data bus) are used to represent a number.

The numbering system we normally use is commonly called the decimal system and is, of course, based on the number 10. There are ten single digit numbers from 0 to 9. This system of numbering is not very convenient for an electronic circuit in that it is difficult to devise a practical system where an output has ten different voltage levels so that any single digit decimal number can be represented. It is much easier to use simple flip/flops which have just two output levels, and can only represent 0 or 1. However, this bars such circuits from operating directly in the decimal numbering system. Instead, the binary system of numbering is utilised.

This system is based on the number 2 rather than 10, and the highest single digit number is 1 rather than 9. If we take a decimal number such as 238, the 8 represents eight units (10 to the power of 0), the 3 represents three tens (10 to the power of 1), and the two represents 2 hundreds (10 to the

power of 2 or 10 squared). Things are similar with a binary number such as 1101. Working from right to left again, the 1 represents the number of units (2 to the power of 0), the 0 represents the number of twos (2 to the power of 1), the next 1 represents the number of fours (2 to the power of 2), and the final 1 represents the number of eights (2 to the power of 3). 1101 in binary is therefore equivalent to 13 in decimal ($1 + 0 + 4 + 8 = 13$).

The table given below shows the number represented by each digit of a 16 bit number when it is set high. Of course, a bit always represents zero when it is set low.

Bit	0	1	2	3	4	5	6
	1	2	4	8	16	32	64

Bit	7	8	9	10	11	12
	128	256	512	1024	2048	4096

Bit	13	14	15
	8192	16384	32768

Using 16 bits any integer from 0 to 65535 can be represented in binary fashion, or using 8 bits any integer from 0 to 255 can be represented, and this exposes the main weakness of the binary numbering system. Numbers of modest magnitude are many binary digits in length, but despite this drawback the ease with which electronic circuits can handle binary numbers makes this system the only practical one at the present time. With its 32 bit registers the 68000 can actually handle very large numbers without difficulty (numbers in excess of 1000 million in fact).

Addition of two binary numbers is a straightforward business which is really more 'simple than decimal addition. A simple example follows:—

First number	11110000
Second number	01010101
Answer	101000101

As with decimal addition, start with the units column and gradually work towards the final column on the left. In this case there is 1 and 0 in the units column, giving a total of 1 in the units column of the answer. In the next column two 0s give 0 in the answer, and the next two columns are equally straightforward. In the fifth one there are two 1s to be added, giving a total of 2. Of course, in binary the figure 2 does not exist, and this should really be thought of as 10 (one 2 and no units), and it is treated in the same way as ten in decimal addition. The 0 is placed in the answer and the 1 is carried forward to the next column of figures. The sixth column again gives a total of 10, and again the 0 is placed in the answer and the 1 is carried forward. In the seventh column this gives a total of 3 in decimal, but in this binary calculation it must be thought of as the binary number 11 (one 2 and one unit). Therefore, 1 is placed in the answer and 1 is carried forward. In the eighth column this gives an answer of 10, and as there are no further columns to be added, both digits are placed in the answer.

Adding two 8 bit binary numbers together produces a slight complication in that, as in this case, the answer is some 9 bits long. When a data register is used to add two numbers it cannot always accommodate the extra bit when there is a final carry-forward, but the 1 in the last column is not simply lost (which would obviously give an incorrect answer and would be unacceptable). Instead, the carry forward is taken to one bit of the microprocessor's status register. Not surprisingly, this is called the carry or "C" register. Like the other status registers this is used to control conditional instructions (i.e. if the carry bit is set high do this, if it is not do that). Anyone who has done some BASIC programming should be familiar with conditional instructions in the form of BASIC IF . . . THEN or IF . . . THEN . . . ELSE instructions. There are other types of instruction which can

produce a carry forward, and these normally activate the carry register (or carry “flag” as it is often termed) when appropriate.

Signed Binary

The binary system described so far, which is often called “direct binary”, is inadequate in many practical applications in that it is unable to handle negative numbers. One way around the problem is to use “signed binary” numbers where the first bit is used to denote whether the number is positive or negative. The convention has the first bit as a 0 for positive numbers and as a 1 for negative numbers. With this system the normal number range of 0 to 255 is replaced with a range of -127 (11111111) to $+127$ (01111111). The problem is solved only at the expense of reduced maximum magnitude for a given number of bits. Note though, that where two or more bytes (or words, or long words) are used to form a long number, only the most significant bit of the high byte needs to be used to indicate whether the number is positive or negative, and it is not necessary to use the most significant bit of each byte in the number to do this.

Obviously a certain amount of care needs to be exercised when dealing with binary numbers and you must know whether a number is in signed or unsigned binary. For example, 10000001 could be 129 (unsigned) or -1 (signed). In this basic form the signed binary system has practical limitations in that it can represent binary numbers without any difficulty, but calculations fail to give the right result, which makes the system of little practical value unless it is modified to correct this anomaly. It is not used with the 68000 microprocessor in the basic form described above.

To illustrate the problem, consider the calculation shown below:—

16	00010000
–5	10000101
Answer (–21)	10010101

Adding 16 and -5 should obviously give an answer of 11 and not -21 .

An alternative and related method of handling negative numbers is the "ones complement" system. Here a negative number is the complement of the positive equivalent. For instance, $+16$ in binary is 00010000, and -16 is therefore 11101111. In other words, the ones are simply changed to zeros and the zeros are changed to ones. This gives better results when used in calculations, as demonstrated by the example given below.

16	00010000
-5	11111010
Answer (266)	100001010

This answer may seem to be less use than the one obtained using ordinary signed binary, and the margin of error is certainly greater, but this depends on how the answer is interpreted. The first point to note is that the positive number starts with a zero and the negative number starts with a 1. Provided that sufficient digits are used this will always be the case, and in this respect the system is not much different to ordinary signed binary. The answer is completely wrong of course, but if the carry is ignored the answer is much closer to the right result. It then becomes 10 rather than 11. So what happens if we try another example and again ignore the carry in the answer?

32	0010000
-4	1111011
Answer (27)	00011011

As before, the answer is wrong, but is one less than the right answer (which is of course 28 in this case).

Twos Complement

Clearly this system can be made to operate properly, and it is just a matter of finding some way of correcting the answer. The standard method used with most microprocessor, including the 68000 is the "twos complement" system. This differs from the ones complement system in that once the complement of a number has been produced, one is added to it. Therefore, rather than -5 being represented as 11111010, it becomes 11111011. If we now apply this to one of the examples given earlier we obtain the following result.

16	00010000
-5	11111011
Answer (11)	00001011

This time, provided we ignore the carry, we have the correct answer of 11. This is a convenient way of handling subtraction (for the microprocessor anyway) since subtraction can be handled by the same circuit that handles addition. To handle a sum such as $45 - 25$ the figure of 25 is converted into (twos complement) -25 , and then added to 45. In other words, rather than calculating the sum in the form $45 - 25$ the microprocessor calculates it as $45 + (-25)$, and either way the answer is 20.

The table given below shows some sample numbers in twos complement form, and should help to clarify the system for you. Note that, like ordinary signed binary, the first digit is used to indicate whether the number is positive or negative.

<i>Number</i>	<i>Positive</i>	<i>Negative</i>
0	00000000	00000000
1	00000001	11111111
2	00000010	11111110
3	00000011	11111101
4	00000100	11111100
32	00100000	11100000

<i>Number</i>	<i>Positive</i>	<i>Negative</i>
126	01111110	10000010
127	01111111	10000001
128	010000000	10000000

Another point to note is that with 8 bit twos complement numbers the range is from +127 to -128.

So far we have only considered calculations where the answer is a positive quantity, but the twos complement system works properly if the answer is negative. The following example demonstrates this point:—

16	00010000
-31	11100001
Answer (-15)	11110001

The system also functions correctly when two negative numbers are added together, as demonstrated by this example:—

-4	11111100
-8	11111000
Answer (-12)	11110100

Overflow Flag

When using the twos complement system there is a slight problem in that a number can be accidentally turned into a negative quantity. The simple calculation shown below demonstrates this point:—

64	01000000
127	01111111
Answer (-65)	10111111

If taken as an ordinary 8 bit direct binary number this does give the right answer, but in the twos complement system the carry forward from bit 6 to bit 7 has changed the sign and magnitude of the number so that an answer of -65 instead of 191 is obtained (we are only dealing with 8 bit numbers here, but the same thing could obviously happen with 16 or 32 bit numbers).

This is termed an "overflow", and it is handled by microprocessors such as the 68000 by a flag called (appropriately) the overflow flag. In the diagram of Figure 1 this is given its abbreviated name, the "V" flag. Like the carry flag, there are special instructions connected with this flag, and these can be used to prevent erroneous results from being produced, or to give warning that an error has occurred. These flags are normally at 0 and are set by an overflow or a carry forward. They are automatically reset by some of the microprocessor's instructions, and this helps to streamline things so that the system operates rapidly and uses as little memory as possible. The 68000 does not have instructions to specifically set or reset flags of the status register.

At this stage it is probably best not to go into any more detail about binary calculations and the way they are handled by microprocessors. It is a complicated subject, and it is probably clarified most easily by trying out a few programs which demonstrate the techniques involved. Some practical examples that can be run on a 68000 based computer loaded with a suitable assembler program are given later in this book. Even if you can only understand direct binary, provided you also understand the main principles of microprocessors you should be able to run and understand some simple assembly language routines.

Binary Coded Decimal

The 68000 can use another form of binary known as "binary coded decimal", or BCD. This is perhaps less frequently used than the twos complement binary system described above, and it has the disadvantages of being relatively slow and uneconomic on memory. However, it can be used to give a high degree

of precision, and it can be advantageous in certain applications. It is certainly worthwhile considering this system briefly here.

With BCD four binary bits (often termed a “nibble”) are used to represent each decimal digit. The system operates in the manner shown below:—

<i>Decimal Number</i>	<i>Bit Code</i>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The binary number is in fact just the normal binary representation of the number concerned, and it is only for numbers of more than 9 that the system is different. The binary codes from 1010 to 1111 are unused, and all two digit decimal numbers require 8 bit binary codes. For instance, the decimal number 64 would be represented by the 8 bit BCD code 01100100. The first four bits (0110) represent the 6, and the second four bits (0100) represent the four. Each byte can therefore represent any two bit number from 0 to 99, which compares to a range of 0 to 255 for a straightforward 8 bit binary number. This helps to contribute to the relative inefficiency of the BCD system. Of course, when a nibble is incremented by 1 from 1001 (9 in decimal) it does not go to 1010 (which is an illegal code in BCD), but cycles back to 0000. A carry forward of 1 should then be taken to the next BCD nibble.

With this system there is no difficulty in handling large numbers, and it is just a matter of using several bytes to accommodate the required number of digits. Negative numbers and decimal points can also be handled with ease by this system, but this requires several additional digits. This

information is usually carried in the most significant bits (i.e. at the left-hand end of the number). Some microprocessors handle BCD by performing calculations in ordinary binary and then adjusting the result using a special instruction. The 68000 performs calculations directly in BCD.

Hexadecimal

While on the subject of numbering systems it would perhaps be worthwhile dealing with another system which you will inevitably come across quite frequently, and this is the hexadecimal system. There is in fact yet another system known as octal which, as its name suggests, is based on the number 8. Octal seems to have fallen from favour in recent years, and as it is something you are not likely to encounter these days we will not consider this system here.

A problem with binary numbers is that they tend to have many digits with each digit being either 0 or 1, which makes them rather difficult to deal with in many circumstances. For instance, dealing with large addresses or microprocessor instruction codes in their binary form would probably be beyond most people's ability. On the other hand, binary numbers give a graphic representation of the state of each bit in the registers of the microprocessor, and this is something that is often important. Decimal numbers are easier to use in that they are much shorter and are in a familiar form. Converting a decimal number into an equivalent binary one is not a very quick and easy process, especially where large numbers are concerned, and this is inconvenient when it is necessary to visualise things on a bit by bit basis.

The hexadecimal system gives the best of both worlds in that it takes just a few digits to represent even quite large numbers, and is in fact slightly better than the decimal system in this respect. On the other hand it is easy to convert hexadecimal to binary, and it is fairly easy to use when operating at bit level. The hexadecimal system is based on the number 16, and there are sixteen single digit numbers. Obviously the numbers we normally use in the decimal system are inadequate for hexadecimal as there are six too few of

them, but this problem is overcome by augmenting them with the first six letters of the alphabet. It is from this that the system derives its name. The table given below helps to explain the way in which the hexadecimal system operates.

<i>Decimal</i>	<i>Hexadecimal</i>	<i>Binary</i>
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	00010000
17	11	00010001
163	A3	10100011

What makes hexadecimal so convenient is the way in which multidigit numbers can be so easily converted into binary form. The reason for this is that each hexadecimal digit represents four binary bits. Take the hexadecimal A3 in the above table for instance. The digit A represents 1010 in binary, and the digit three converts to 0011. A3 therefore represents 10100011 in binary. You may find that you can memorise the four bit binary number represented by each of the sixteen hexadecimal digits, but a little mental arithmetic is all that is needed to make the conversion if you cannot.

The digits in a hexadecimal number represent, working from right to left, the number of units, 16s, 256s, 4096s, 65536s, 1048576s, 16777216s and 268435450s (approx.).

You are unlikely to use hexadecimal numbers of more than eight digits in length, and mostly you will deal with hexadecimal numbers of only five digits or less..

System Operation

If we now return to the block diagrams of Figures 1 and 2, you should begin to get the idea of how data is moved around the system and processed. At switch-on the microprocessor has all the registers set to zero, including the program counter which uses the numbers in the bytes at the bottom of the address range to provide the start address of the computers operating system program. This start-up procedure is not normally of interest to the machine code or assembly language programmer, since few people design their own systems, and it is certainly only of academic interest to anyone writing assembly language programs for a home or personal computer. The program you write will normally go into a section of memory occupied by random access memory (RAM). This is memory where the microprocessor can set its contents at any desired 8 or 16 bit binary number (depending on the version of the 68000 in use), and then read back that number at a later time. The contents of RAM can be changed an unlimited number of times, but reading the contents of RAM does not destroy the data there or affect it in any way. However, when the computer is switched off the contents of RAM are lost.

Software such as the computer's operating system and BASIC interpreter are usually in read only memory (ROM) which retains its contents after the computer has been switched off (although the BASIC interpreter or other language does actually have to be loaded from tape or disc on a few home computers, and this is quite common with personal computers). The contents of ROM are fixed, and writing to ROM does not alter its contents. ROM is not an area of memory that is normally used by the programmer, the exception being when there are useful routines there that can be utilized.

The block marked input/output in Figure 2 includes such things as the keyboard and the chip which produces the

television picture. The 68000 series of microprocessors all use memory mapped input/output. In other words, the microprocessor reads data from or writes data to input/output devices just as if they were RAM, and they are addressed in exactly the same way. This has the advantage of making programming more straightforward in that using a common set of instructions for memory and input/output operations gives fewer instructions to contend with. A minor drawback is that some of the memory address range is occupied by the input/output devices, but this does not normally seriously deplete the maximum amount of memory that can be included. This is especially so in the case of the 68000 with its massive address range of over 16000000, and is hardly likely to be a problem with the 68008 and its address range of over 1000000. Incidentally, when dealing with memory it is common for the term "k" to be used, and this refers to the kilo-byte of memory, which is actually 1024 bytes. A term that is often used with the 68000 series and other advanced microprocessors is "M", or mega-bytes of memory. This is 1048576 bytes or words of memory, and the 68000 therefore has a 16M address range.

With the aid of the computer's operating system and either the BASIC interpreter or an assembler, the machine code program is placed in a suitable section of memory, and the program is run by directing the microprocessor to the appropriate address using the appropriate instruction. Exactly how the program is run depends on whether it is true machine code or assembly language. With the QL computer for example, machine code is run from its Super BASIC using the CALL instruction which can include data to be placed in some of the microprocessor's registers, as well as the start address of the routine. With assembly language things are dependent upon the exact facilities that the assembler provides, but the subject of assemblers is something that will not be considered in more detail until the final section of this book.

Once into the machine code or assembly language program, by whatever means, the program then operates by fetching an instruction from the specified start address of the program, and then shuffling data around its registers and the memory as it goes through the set of instructions. This may seem a

rather vague description of things, but if you can grasp the basic concept of instructions and data being taken from memory, or possibly input/output devices, with the data being processed in some way by the microprocessor before being sent back to a memory location or an output device, then you should not find it difficult to understand a few simple machine code programs and then gradually progress to more complex ones. If you cannot see how the system operates overall, individual machine code instructions could, to say the least, be rather difficult to understand, and even simple programs would certainly be impossible to follow.

A simple example of how the system operates should now be quite easy for you to understand. We will assume that the program must take a number from one memory location, then add this to a number taken from a second address, and then finally place the answer at a third address. There is more than one way of going about this, and the differences occur due to the various addressing modes that the 68000 can use. In other words, we can place the numbers at virtually any addresses we like, and by using the appropriate addressing mode (or modes) and instructions the program can be made to obtain the numbers from the correct addresses. Addressing modes is a fairly complex subject which is considered in detail in a later chapter of this book, and it will not be considered any further here. A simple addition program could run along the following lines:—

```
Move immediate data to register D0
Add immediate data to register D0
Move immediate data to register A0
Move contents of D0 to address stored in A0
Return from subroutine
```

The first instruction takes the data in the byte or bytes of memory immediately following the instruction and moves it to data register D0. This is known as immediate addressing and is the most simple type. The data following the instruction is the first number in the calculation. The next instruction adds the immediate data (the second number in the calculation) to the contents of data register 0. The next

instruction loads address register A0 with the immediate data, which in this case is the address where the result of the calculation is to be stored. The next instruction moves the contents of D0 (the result of the addition calculation) to the address stored in A0. Finally, the return from subroutine instruction hands control back to the computer's operating system, and terminates the program in a way that will not crash the computer.

Something that will probably have become apparent is that it takes a large number of machine code or assembly language instructions to achieve quite simple tasks. When programming in a language such as BASIC each instruction is converted into a number of machine code instructions by the interpreter. This is one of the factors which makes writing machine code programs a relatively slow affair.

Something that is less apparent is that most instructions have a large number of variations. The move instruction for example, can move data from practically any desired location to virtually any desired destination with a variety of addressing modes to choose from. It is also necessary to specify whether the data is a byte, word, or long word. When using an assembler things are relatively straightforward, with an instruction such as: `MOVE D0,(A0)` being used. This simply moves the contents of register D0 to the memory location addressed by register A0. When using machine code it is necessary to work out the state of each bit in the code number, choosing the states that give not just the required instruction but also the required variation, and then enter this into the computer. This helps to make machine code programming a very difficult and long process with any 68000 series microprocessor, and is the reason that the use of an assembler has to be recommended for anyone writing more than just a few very short machine code routines.

It should perhaps be explained that although the address registers are 32 bit types, 8 bits are left unused when they are used to hold 24 bit addresses (or 12 bits are left unused in the case of the 68008 with its 20 bit address bus). It is bits 24 to 31 (or 20 to 31 for the 68008) that are left unused. Similarly, if a data register is used to hold a word bits 16 to 31 are not used, and bits 8 to 31 are not utilized if a data

register is used to hold a byte.

The Stack

There are a couple of registers in the 68000 (and shown in Figure 1) which we have not yet considered, and we will take a look at the function of these now. These are the two stack pointers (address register A7), one of which is used in the user mode while the other is utilized in the supervisor mode. The idea of these two modes is for the supervisor one to be used by the operating system, and the user mode to be used for any other programs. A few of the 68000 instructions are privileged and are only available in the supervisor mode. There are output terminals on the microprocessor which indicate its operating mode and can be used to disable hardware in the user mode so as to restrict user access to areas of memory or input/output devices (restrict rather than prevent access since it is usually possible to call operating system routines which make use of the protected memory or other hardware). This is a subject with which you do not really need to become deeply involved at this stage. It is acceptable for the two stack pointers to have the same register number (A7) since only one or the other can be used at one time, depending on which operating mode the microprocessor is set to.

So far we have only considered the reason for having two stack pointers, and not the purpose of these and the stack itself. The purpose of the stack is to act as a convenient place for temporary data storage, and some microprocessors have the stack as an internal part of the microprocessor. This is often termed a "hardware stack". This is in many ways the most elegant solution to the problem, and it has the advantage of high speed. It has the disadvantage of giving only a relatively small number of registers, and does of course add complexity to the microprocessor.

The 68000, in common with most general purpose microprocessors, uses the alternative of a "software stack". This is just an area of memory which is reserved for use as the stack, and the system must, of course, provide RAM at the relevant range of addresses. The stack pointer points to an address in

this block of RAM, and with the 68000 the stack can be any section of memory that provides RAM. The stack uses the "last in — first out" or LIFO system. In other words, each time data is placed onto the stack the stack pointer is incremented by 1, and each time data is taken from the stack the pointer is automatically decremented by one. This is often looked on as being analogous to a stack of plates, with plates being loaded one on top of the other, building a pile from the bottom upwards, and then removing plates from the top of the pile and working downwards. This analogy does not work too well with the 68000 as its stack pointer counts downwards as the stack is enlarged. However, the last in — first out doctrine still applies. The fact that the stack grows downwards is really only of academic importance anyway, since the stack pointer increments and decrements automatically.

Apart from use as a convenient temporary data store, the stack is also used when subroutines and interrupts are implemented. We will not consider these in detail here, but in both cases the microprocessor breaks out of its normal operating routine, and branches off into another routine. With an interrupt the signal to the microprocessor that it must break out of its normal routine is provided by a hardware device via the 68000's three interrupt inputs. A typical application where interrupts are used is the timer that is a feature of many home-computers. Here a counter circuit generates an interrupt (say) every 10 milliseconds, and a software routine is used to increment by one the number stored at a set of memory locations. With suitable manipulation the number in these RAM locations can be converted into suitable data for a minutes and seconds display, or even for a real-time clock. The number can be POKEd to any desired figure so that the "clock" can be set at the required time. If the timer is to achieve a reasonable degree of accuracy it is important that the microprocessor carries out the software routine at each request without waiting to complete other tasks first. It is for this type of application that interrupts are ideal.

The problem with the use of interrupts is that the microprocessor has to be able to break back into its main routine again after it has finished the interrupt routine. To facilitate

this it is necessary to store on the stack the contents of any registers that are in use when the interrupt is generated. After the interrupt has been serviced the data on the stack is returned to the registers from where it was obtained so that the program can continue from where it left off. The general situation is the same when a subroutine is called, and a subroutine could be regarded as a software generated interrupt.

When writing programs for home-computers it is unlikely that you will need to deal with interrupts, and they are principally used as part of the computer's operating system and in a few specialised add-on hardware applications. A detailed description of the 68000's sophisticated system of interrupts would certainly be out of place in an introductory book such as this one. However, it is important to realise that interrupts are occurring while machine code programs are running, since in some applications the disruption caused by interrupts can prevent proper operation of the program. This occurs where timing of the program is critical, and the delays produced by interrupts prevent the computer from providing some event at strictly regular intervals. When a program of this type is running it is necessary to disable interrupts.

Flags

The 68000 has status flags apart from the carry and overflow flags, and one of these is the "zero" bit. This is used by conditional instructions which test to see whether or not this bit is set. As its name suggests, this bit is set when the result of an operation by the arithmetic logic unit has produced zero as the answer.

The negative flag or N bit is equally straightforward, and this bit is set when the result of an operation by the arithmetic logic unit gives a negative result.

There are three interrupt mask bits in the status register, and these are used to determine which devices connected to the interrupt inputs are able to generate interrupts and which are not. As mentioned earlier, there are three interrupt inputs on the 68000, and the idea is for each device connected to these inputs to produce a different binary pattern on them.

The lower the number, the higher the priority of the interrupting device. Most microprocessors have a system of interrupts that enables devices which must be serviced most urgently to take precedence over less important devices, but the seven tier system of the 68000 is far superior to most other microprocessors. Note that only seven and not eight different levels of interrupt priority are available since there must be an inactive state.

The number placed in the interrupt mask bits enables devices having an interrupt priority level equal to or less than that number to generate interrupts. For instance, placing 101 (5 in decimal) in the interrupt mask bits enables devices of priority level 5 or less to generate interrupts. Placing all three masking bits at zero therefore disables interrupts.

The "S" bit of the status register selects the operating mode, and is set at 1 in the supervisor mode and 0 in the user mode.

The "X" or extend bit is a form of carry flag which is used in microprecision arithmetic, and is something which goes beyond the scope of this book.

An advanced feature of the 68000 is its trace mode, and the device is forced into this mode by setting the "T" bit or trace flag to 1. This is again something which goes beyond the scope of this book, but the idea of the trace mode is to enable the microprocessor to go through a program one instruction at a time, enabling the contents of the registers to be examined between instructions. This is something which aids the debugging of programs, and which can be achieved with virtually any microprocessor using a suitable monitor program, but it is something which is more easily implemented with the 68000 and its trace mode.

Chapter 2

ADDRESSING MODES

Addressing is the means by which the processor determines the location of the data, or operand, on which the instruction is to operate. Some instructions (for example, MOVE) require two addresses, the source location and the destination location.

Most instructions can use more than one addressing mode. Where two addresses are required, a different mode of addressing may be used for each.

The MC68000 has 11 addressing modes.

1. Implicit

In fact, these instructions require no operand as such, though they may store or retrieve data from the stacks. Examples are NOP and RESET, which require no operand at all, and RTE, RTS, RTV and the TRAP instructions, which use one or other of the stacks.

Also in this category are those instructions which make implied reference to one of the 68000 registers. These include Branch and Jump instructions (which alter the program counter), and some MOVE instructions which alter specific registers.

In general, instructions which use implicit addressing can *only* use implicit addressing.

2. Register Addressing

This mode is used to specify an operand residing in one of the 68000's internal registers. It is subdivided into two self-explanatory modes, Data Register Direct, and Address Register Direct.

Most 68000 instructions allow Register addressing, but there are a few which can only use registers as operands.

These are EXG (exchange registers), EXT (sign extend), and SWAP (swap register contents). Certain MOVE instructions can also only use registers as operands.

There are also some instructions which require that one of the operands be specifically either an address or a data register. For example, the ADD instruction requires one operand to be a data register, and the ADDA instruction requires one operand to be an address register.

3. Immediate Addressing

In this mode, the data follows immediately after the opcode within the program in memory. This addressing mode is used to include constant data within programs.

The 68000 has instructions for immediate data which can be of byte, word (two byte), or long word (four byte) length. This is indicated by appending B, W, or L to the instruction, for example ADD.L or SUB.B. Word length is assumed if no indication is given.

The 68000 also has a 'quick' immediate addressing mode for small operands. In these, the data is actually contained within the opcode. ADDQ and SUBQ can add and subtract numbers from 1 to 8. MOVEQ can be used to move numbers from 0 to 255 (or -128 to +127) in 2's complement) to a register or memory location.

In assembly language, immediate addressing is indicated by preceding the data with a 'hash' sign (#), thus:—

MOVE #2000,D3

This instruction would load 2000 (decimal) into the register D3.

4. Absolute Short Addressing

Absolute addressing means that the address of the data follows immediately after the opcode in memory.

In absolute short addressing, the low-order half of the address follows directly after the opcode in memory. The

high-order half of the address is assumed to be either 0000 or FFFF (hex.), depending on whether bit 15 is 0 or 1. This means if the two bytes following the opcode contain 0000 to 7FFF, this will be the address of the data, but if they contain 8000 to FFFF, the addresses accessed will be from FFFF8000 to FFFFFFFF. Thus this mode can address the top 32K bytes of memory, and the bottom 32K. This is called 'sign extension', because bit 15 would be the sign bit in 2's complement representation.

It should be noted that the 68008 processor used in the Sinclair QL can only address memory locations up to FFFFF hex. Also, the bottom 32K of memory in the QL is occupied by the system ROM.

This mode of addressing will be used automatically by an assembler when appropriate. It saves memory space and time compared to absolute long addressing.

5. Absolute Long Addressing

In this mode, the four bytes after the opcode contain the addressing where the data is to be found. This mode allows the processor to access any byte in memory, though it is not used where absolute short addressing can be used. Again, it must be remembered that the 68008 can only use addresses up to FFFFF hex.

Address Register Indirect Addressing

In indirect addressing, an address in memory follows the opcode, and the contents of this memory location are the address from which the data must be fetched. The 68000 series microprocessors do not provide true indirect addressing (in fact very few processors do), but they do allow the address of the data to be held in one of the processor's address registers.

In assembly language, register indirect mode is indicated by enclosing the address register to be used in brackets, thus:--

MOVE D1,(A5)

This instruction means 'move the contents of register D1 to the memory location addressed by A5'.

6. Address Register Indirect with Displacement

This is similar to address register indirect, but the address in the register is modified by an offset which follows the opcode in program memory, and is therefore normally a constant offset. This mode is useful when we want to access a particular item in an array or list. The base address of the array is contained in the address register, and the various elements can be accessed by the offsets following the instructions.

In assembly language, this mode is indicated by placing the offset before the address register (which is enclosed in brackets, as for register indirect), thus:—

MOVE 40(A5),D0

This instruction means 'move the contents of the memory location in register A5+40 bytes to register D0'.

Note that this mode does not alter the contents of the address register.

7. Address Register Indirect with Index and Displacement

This is a further extension of the preceding mode, the address of the data being the sum of three elements; the contents of an address register, the contents of an index register (which can be any data or address register), and a displacement included in the instruction.

This is used for similar purposes to the preceding mode, but it is more useful for structured data. The address register can contain the base address of the data, the index register can be used to point to a particular record in the data, and the displacement to a particular part of the record.

The assembly language form is a fairly obvious extension of the foregoing modes, with the address register and the index

register included together in the brackets, thus:—

MOVE 40(A0,A1),D0

This means 'move the contents of memory location (A0+A1+40) to D0.

8. Address Register Indirect with Postincrement

This mode is used when accessing successive bytes in memory, as for example when printing a string or moving a block of memory, or when accessing successive elements in an array of data.

The address in the base register can be incremented by one, two, or four bytes after each operation, depending on whether the byte, word or long word form of the instruction is in use.

This mode is indicated in assembly language by placing a ' + ' sign after the closing bracket around the address register, thus:—

MOVE.B (A1)+,D1

This instruction means 'load the contents of the memory location addressed by register A1 into register D1, then increment A1 by 1'.

9. Address Register Indirect with Predecrement

This is essentially similar to the foregoing, but is used to move backwards through memory rather than forwards. This may be preferred for some operations, and is essential for moving a block of memory downwards (i.e. to lower addresses) when the new addresses overlap the old.

In this case, the contents of the address register are modified before the operation is carried out rather than after. This order of postincrement but predecrement is used to maintain consistency with the way in which the 68000 stack pointers operate.

This mode is indicated in assembly language by putting a ' -- ' sign *before* the brackets enclosing the address register, thus:—

MOVE.L -(A1),D1

This instruction means 'decrement the contents of register A1 by 4 (long word form) then move the contents of the memory location addressed by A1 to register D1'.

10. Program Counter Relative with Displacement

This is really a special case of register indirect addressing, but using the program counter rather than one of the address registers. This means that a program can run correctly wherever it is loaded in memory, because all addresses can be specified relative to the current position in the program. In other words, it is an aid to writing 'position independent code'.

This mode is similar to register indirect with displacement, in that the instruction contains a constant displacement after the opcode.

The assembly language form is also similar, but with 'PC' in place of an address register in the brackets, thus:—

MOVE 20(PC),D2

11. Program Counter Relative with Index and Displacement

This is an extension to the previous mode, and is provided for the same reasons. It is essentially similar to register indirect with index and displacement, but using the program counter in place of an address register.

The index register can be any of the address or data registers.

The allowable assembly language forms for this mode vary somewhat between different assemblers, and it would probably not be helpful to give examples here. The documentation with an assembler should include this information.

Chapter 3

THE INSTRUCTION SET

In this chapter, all the 68000 instructions are described. The address modes available for each instruction are given in chart form, except where only one mode is available (mostly instructions using implied addressing).

Charts of opcodes are not given. The way in which the registers, for instance, are coded within instruction codes, makes the opcodes very tricky to calculate, even for experts. Hand assembling for 68000-series microprocessors is simply not a practical proposition.

Where the result of an operation has to be stored, it is to be assumed that it is stored in the destination data location unless otherwise indicated.

ADD BINARY CODED DECIMAL

Mnemonic – ABCD

This instruction adds the source data and the value of the extend flag X to the destination data, using binary coded decimal (BCD) arithmetic.

This instruction can use either register direct addressing or address register indirect addressing, but both operands must use the same mode. It can therefore be used for register-to-register addition, or memory-to-memory addition. Only the least significant 8 bits of the data are affected.

When address register indirect addressing is used, the addresses in both registers are decremented before the addition. This is to simplify multi-byte BCD arithmetic. Strings of BCD digits are normally stored with the least significant digit at the highest memory address.

The carry (C) and extend (X) flags are set if a decimal carry is generated, and are cleared otherwise. The zero (Z) flag is cleared if the result is non-zero, unchanged otherwise. Note this: the zero flag is not changed if the result is zero. In multiple precision arithmetic, the zero flag should be set

first. If any part of the result is non-zero, the flag will be cleared, otherwise it will remain set. The N and V flags are undefined.

ADD BINARY

Mnemonic — ADD

This instruction adds the source data to the destination data, storing the result in the destination. One of the operands for this instruction must be a data register. All address modes except implied can (in general) be used for the other operand, but it must be remembered that address registers cannot handle byte-size data (see charts).

The C and X flags are set if a carry is generated and cleared otherwise. The Z flag is set if the result is zero and cleared otherwise. The N flag is set if the result is negative and cleared otherwise. The V flag is set if an overflow occurs and is cleared otherwise.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct	x	
Address register indirect	x	
Postincrement register indirect	x	

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

ADD ADDRESS

Mnemonic – ADDA

This is a special form of the ADD instruction to allow a source operand to be added to a specific address register. All address modes are allowed for the source operand. Note that this instruction *does not* affect *any* of the status flags.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	
Address register direct	x	x
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

ADD IMMEDIATE

Mnemonic - ADDI

This instruction performs decimal addition between immediate data and the source operand.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

The effect on status flags is the same as for ADD.

ADD QUICK

Mnemonic – ADDQ

This is a special form of immediate-addressing addition for small source operands (i.e. 1 to 8). This form has the advantage of being faster and using less memory space.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		x
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

The status flags are affected in the same way as for the ADD instruction.

ADD EXTENDED

Mnemonic – ADDX

Adds the source data plus the value of the X flag to the destination data. As with ABCD, this instruction has register-to-register and memory-to-memory forms. The data size can be byte, word, or long word. It is used in multiple-precision arithmetic.

In the address register indirect (memory to memory) form, the contents of both address registers are automatically decremented before the addition is performed (by 1, 2, or 4 depending on whether the B , W , or L form of the instruction is specified).

The C and X flags are set if a carry is generated, cleared otherwise. The Z flag is clear if the result is non-zero, otherwise unchanged (see ABCD for comment on this). The N flag is set if the result is zero and cleared otherwise. The V flag is set if there is an overflow, cleared otherwise.

LOGICAL AND

Mnemonic – AND

This instruction performs a logical bitwise AND between the source data and the destination data. One of the data registers must be used as either the source or the destination. In general, most other address modes can be used for the other operand, as in the charts.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
P.C. relative with index		
Immediate		

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct	x	
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

The N flag will be set if the most significant bit of the result is 1, cleared otherwise. The Z flag will be set if the result is zero, cleared otherwise. The V and C flags are *always* cleared, and the X flag is unaffected.

AND IMMEDIATE

Mnemonic — ANDI

Performs logical bitwise AND between immediate data and the destination operand. The chart shows the available address modes.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		x
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

In addition, the destination operand may be the condition codes, or the entire status register. If it is the entire status register, this is a privileged instruction and can *only be executed in Supervisor Mode*.

This instruction may be specified as byte, word, or long word. The immediate data supplied must match the operand size specified, that is, one or two words. If a byte operand is specified, the second (low-order) byte is used.

The effect on the status flags is the same as for AND, except, of course, where the status register is the destination. In this case, the flags are set by the result of the operation.

ARITHMETIC SHIFT LEFT (DATA REGISTER)

Mnemonic – ASL

This instruction shifts the contents of a data register to the left. The bit shifted out is placed in the C and X flags, and zeroes are moved in on the right.

Shifting by more than one bit position is allowed. The shift count may be specified by another data register, or immediate data. In the first case, the shift may be from 0 to 63 places, in the latter from 1 to 8.

The C and X flags receive the shifted-out bits. If the shift count is zero, C is cleared and X is unaffected. The V flag indicates any sign change during the operation. The N and Z flags are altered depending on the value of the result.

ARITHMETIC SHIFT LEFT (MEMORY)

Mnemonic – ASL

Similar to the previous instruction, but operates on an operand in memory, is restricted to word-length data, and

can shift by one position only.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect		X
Postincrement register indirect		X
Predecrement register indirect		X
Register indirect with offset		X
Register indirect with index		X
Absolute short		X
Absolute long		X
P.C. relative with offset		
P.C. relative with index		
Immediate		

The C and X flags are set by bit 15 of the (original) data, the V flag indicates any sign change.

ARITHMETIC SHIFT RIGHT

Mnemonic – ASR

This instruction shifts the bits of the operand to the right. The bits shifted out to the right go into the C and X flags, and zeroes are moved in on the left.

As with ASL, there are versions of this instruction for operations on data registers and memory locations, and the same restrictions and flag indications apply, as do the address modes applicable.

BRANCH ON CONDITION

This group of instructions test the flags, and may cause the program to branch forward or back depending on the results. Thus they are the decision-making instructions.

These instructions are most frequently preceded by a compare instruction, but they may follow any instruction which conditions the flags.

BHI — Branch high

Branches if both the C and Z flags are clear.

BLS — Branch Low or Same

Branches if either the C or Z flag is set.

BCC — Branch if Carry Clear

Self-explanatory.

BCS — Branch if Carry Set

Self-explanatory.

BNE — Branch if Not Equal (to zero)

Branches if the Z flag is clear.

BEQ — Branch if Equal (to zero)

Branches if the Z flag is set.

BVC — Branch if oVerflow Clear

Self-explanatory.

BVS — Branch if oVerflow Set

Self-explanatory.

BPL — Branch if PLus (positive)

Branches if the N flag is clear.

BMI — Branch if MInus (negative)

Branches if the N flag is set.

BGE — Branch if Greater or Equal

Branches if the N and V flags are either both set or both clear.

BLT — Branch if Less Than

Branches if N is set and V is clear, or if N is clear and V is set.

BGT — Branch if Greater Than

Branches if N and V are set and Z is clear, or if N and V and Z are all clear.

BLE – Branch if Less or Equal

Branches if Z is set, or if N is set and V is clear, or if N is clear and V is set.

The only address mode used by these instructions is program counter relative. The displacement from the instruction location can be from -126 to +129, or from -32766 to +32769. Note that these figures represent memory locations, not instructions.

The displacement is added to the program counter after the program counter has been incremented by two.

Branch instructions do not affect the flags. The old value of the program counter is lost.

TEST A BIT AND CHANGE

Mnemonic – BCHG

This instruction tests the state of a bit in a memory location or data register, and complements it. The original state of the bit is reflected in the Z flag. The number of the bit to be tested may be either immediate data, or held in a data register. Allowable address modes for the destination data are shown in the chart.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

If the bit to be tested is in memory, then BCHG is restricted to byte-size data.

This instruction only affects the Z flag.

TEST A BIT AND CLEAR

Mnemonic – BCLR

This instruction is identical to BCHG, with the exception that the tested bit is always set to zero, regardless of its original condition. For allowable address modes, etc., see BCHG.

Again, only the Z flag is affected.

BRANCH ALWAYS

Mnemonic – BRA

This instruction is in all essentials similar to the conditional branches, except that no testing of flags occurs, and the program always branches. It is thus exactly equivalent to the BASIC GOTO. For information on the range of allowable displacements, etc., see the section on conditional branches.

TEST A BIT AND SET

Mnemonic – BSET

This instruction is identical to BCHG and BCLR, except that the tested bit is always set. See BCGH for allowable address modes, etc.

BRANCH TO SUBROUTINE

Mnemonic – BSR

This instruction is similar to BRA, but before branching it saves the address of the instruction following BSR on the stack. Thus an RTS (return from subroutine) instruction at the end of the subroutine allows program execution to continue with the instruction after BSR.

This instruction allows a relative unconditional jump to a subroutine, and is useful when writing position-independent code.

No flags are affected.

TEST A BIT

Mnemonic – BTST

This belongs to the same family as BCHG, BCLR, and BSET, but in this case, the tested bit is not altered. See BCHG for other information.

CHECK REGISTER AGAINST BOUNDARIES

Mnemonic – CHK

This instruction checks the contents of a data register against a source operand. If the contents of the data register are less than zero or greater than the source operand, a TRAP is generated, and the processor enters exception processing. Only the least significant 16 bits of the data register are used. There are no byte or long word versions of this instruction.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

This instruction provides a simple means of checking that a block of data has not exceeded the space allowed for it.

The N flag is set if the contents of the data register are less than zero, and cleared if the source data is less than the contents of the data register. The C , V , and Z flags are affected, but are left undefined. The X flag is not affected.

CLEAR AN OPERAND

Mnemonic – CLR

This instruction sets a data register or memory location to zero. The data size for this instruction can be B , W , or L . W is assumed if length is not specified.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

This instruction always sets the Z flag, and clears the N , V , and C flags. The X flag is not affected.

COMPARE

Mnemonic – CMP

This instruction subtracts the contents of a memory location or data register from the contents of a data register. The result of the subtraction is not stored, but the status flags are set according to the results. Both operands are unchanged.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct	x	
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

Note that address register direct addressing cannot be used if the data size is byte.

The C , N , V , and Z flags are all affected by this operation, the C flag representing a borrow. X is not affected.

Compare instructions are normally followed by a conditional branch.

COMPARE ADDRESS

Mnemonic – CMPA

This is a special version of the CMP instruction using an address rather than a data register as destination operand. Apart from this, and the fact that it can only use word and long-word size operands, it is equivalent to CMP.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	
Address register direct	x	x
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

For the effect on flags, see CMP.

COMPARE IMMEDIATE

Mnemonic – CMPI

This instruction subtracts the immediate data following the opcode from the destination data. The result is discarded, but the status flags are set according to the result of the subtraction. The destination data is unaltered.

This instruction can be specified as B , W , or L size, and the immediate data supplied must match the size specified. If a byte operand is specified, the low-order (second) byte of the immediate data is used.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

For the effect on the status flags, see CMP.

COMPARE MEMORY

Mnemonic – CMPM

This instruction compares the contents of two memory locations, the addresses of which are held in two address registers. This instruction can only use postincrement register indirect addressing. As with other compare instructions, the contents of the memory locations are not altered, but the status flags are modified.

This instruction is useful for searching a block of memory for occurrences of a particular value, starting at the lowest address and continuing to the highest, branching every time an occurrence is found.

The effect on the status flags is the same as for CMP.

TEST CONDITION, DECREMENT, AND BRANCH

This group of instructions are conditional branches which test both the status flags and a data register.

The status flags are tested first. If the condition is met, the next instruction in sequence is executed. If it is not met, the data register contents is decremented by 1. Then the contents of the data register are tested. If it contains -1, the next instruction in sequence is executed. If not, the program branches.

A 16-bit displacement is specified, so the program can branch by from -32766 to +32769 bytes (not instructions).

DBT – True

Branches if true (1).

DBF – False

Branches if false (0).

DBHI – High

Branches if both the C and Z flags are clear.

DBLS – Low or Same

Branches if either the C or Z flag is set.

DBCC – Carry Clear

Self-explanatory.

DBCS – Carry Set

Self-explanatory.

DBNE – Not Equal (to zero)

Branches if the Z flag is clear.

DBEQ – Equal (to zero)

Branches if the Z flag is set.

DBVC – oVerflow Clear

Self-explanatory.

DBVS – oVerflow Set

Self-explanatory.

DBPL – PLus (positive)

Branches if the N flag is clear.

DBMI – MInus (negative)

Branches if the N flag is set.

DBGE – Greater or Equal

Branches if the N and V flags are either both set or both clear.

DBLT – Less Than

Branches if N is set and V is clear, or if N is clear and V is set.

DBGT – Greater Than

Branches if N and V are set and Z is clear, or if N and V and Z are all clear.

DBLE – Less or Equal

Branches if Z is set, or if N is set and V is clear, or if N is clear and V is set.

This instruction is very valuable in implementing repetitive loops, using a data register as a loop counter, as the instruction both tests a condition and decrements the loop count.

This instruction does not affect the status flags.

DIVISION (SIGNED)

Mnemonic – DIVS

This instruction divides the 32-bit contents of a data register (destination operand) by a 16-bit source operand. The division is performed using 2's complement binary arithmetic. A 32-bit result is obtained, consisting of the quotient in the least 16 bits of the destination data register, and the remainder in the most significant 16 bits. The sign of the remainder and the sign of the dividend unless the remainder is zero.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

The **C** flag is always cleared. The **N** flag is set if the quotient is negative and cleared otherwise, but undefined if an overflow occurs. The **V** flag is set if the source operand is larger than the destination operand (an overflow), and the division is aborted, leaving the operands unchanged. The extend flag **X** is not affected. The **Z** flag is set if the quotient is zero, cleared otherwise.

If division by zero (a mathematical paradox) is attempted, the instruction is aborted, and a **TRAP** is generated. The processor will automatically enter exception processing.

DIVISION (UNSIGNED)

Mnemonic – DIVU

This instruction divides the 32-bit contents of a data register (destination operand) by a 16-bit source operand. The division is performed using unsigned binary arithmetic. A 32-bit result is obtained, consisting of the quotient in the least significant 16 bits, and the remainder in the most significant 16 bits.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

Apart from the **N** flag, the flags are set as for the **DIVS** instruction. The **N** flag is set if the most significant bit of the quotient is set, cleared otherwise. If an overflow occurs, the **N** flag is undefined.

The effect of an attempted division by zero is the same as with the **DIVS** instruction.

LOGICAL EXCLUSIVE-OR

Mnemonic – EOR

This instruction performs a bitwise exclusive-OR of the contents of a data register with the contents of the destination operand.

EOR is performed according to the following rules:–

1 EOR 1 = 0

1 EOR 0 = 1

0 EOR 0 = 0

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

The N flag is set if the M.S.B. of the result is set. The Z flag is set if the result is zero, cleared otherwise. V and C are always cleared, and X is not affected.

EXCLUSIVE OR IMMEDIATE

Mnemonic – EORI

Performs the EOR operation between immediate data and the destination operand. See EOR for logic rules.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

In addition, the destination operand may be the condition codes or the entire status register. When it is the entire status register, the instruction is *Privileged*, and may only be executed while the processor is in supervisor mode.

The effect on the flags is the same as EOR.

EXCHANGE REGISTERS

Mnemonic – EXG

This instruction swaps over the contents of two 32-bit registers. Only register direct addressing is appropriate.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct	x	x
Address register indirect		
Postincrement register indirect		
Predecrement register indirect		
Register indirect with offset		
Register indirect with index		
Absolute short		
Absolute long		
P.C. relative with offset		
P.C. relative with index		
Immediate		

Note that the order in which the registers are given is immaterial – EXG A4,A6 is the same as EXG A6,A4.

The status flags are not affected.

SIGN EXTEND

Mnemonic – EXT

This instruction extends bit 7 (word-length instruction) through bits 8–15, or bit 15 (long-word instruction) through bits 16–31 of a data register.

Only data register direct addressing is appropriate.

The **N** flag is set if the result is negative, cleared otherwise. The **Z** flag is set if the result is zero, cleared otherwise. **C** and **V** are always cleared, and **X** is unaffected.

JUMP

Mnemonic – JMP

This instruction loads a new address into the program counter, causing an unconditional jump to that address. The old address is lost.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect		x
Postincrement register indirect		
Predecrement register indirect		
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		x
P.C. relative with index		x
Immediate		

No flags are affected.

JUMP TO SUBROUTINE

Mnemonic – JSR

This instruction causes an unconditional jump to a new address, but saves the old address on the stack. This

instruction is similar to BSR but uses an absolute address rather than a relative displacement.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect		x
Postincrement register indirect		
Predecrement register indirect		
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		x
P.C. relative with index		x
Immediate		

No flags are affected.

LOAD EFFECTIVE ADDRESS

Mnemonic – LEA

This instruction calculates an effective address, and stores it in one of the address registers.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		x
Address register indirect	x	
Postincrement register indirect		
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate		

No flags are affected.

LINK AND ALLOCATE

Mnemonic – LINK

This instruction saves the contents of a specified address register on the stack, and then stores the (updated) stack pointer in that register. A sign-extended 2's complement displacement from the instruction is then added to the stack pointer.

The purpose of this instruction is to allow space (called a 'frame') to be reserved on the stack for local variables in a subroutine. The UNLINK instruction is used at the end of the subroutine to clear up the stack.

Note that the displacement must be a negative number to reserve space. A positive displacement will cause the stack to be overwritten.

No flags are affected.

LOGICAL SHIFT LEFT (DATA REGISTER)

Mnemonic – LSL

This instruction has the same effect as ASL (data register). See that instruction for details.

LOGICAL SHIFT LEFT (MEMORY)

Mnemonic – LSL

This instruction has the same effect as ASL (memory). See that instruction for details.

LOGICAL SHIFT RIGHT

Mnemonic – LSR

This instruction is identical to ASR, except that zeroes are moved into the most significant bit position, instead of it being kept intact.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

Consult the ASR instruction for details of LSR.

The effect on the flags is the same as for ASR, except that the N flag is always cleared (because the MSB is always cleared).

MOVE DATA

Mnemonic – MOVE

This instruction is used to move data. There are very few restrictions on address modes with this instruction. It is the equivalent of the LOAD and STORE instructions provided on other processors, but data can be moved from memory to memory without intermediate use of a processor register.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct	x	
Address register indirect	x	x
Postincrement register indirect	x	x
Predecrement register indirect	x	x
Register indirect with offset	x	x
Register indirect with index	x	x
Absolute short	x	x
Absolute long	x	x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

Address register direct mode is not valid if the data size is byte.

The data is examined as it is moved, and the N and Z flags are conditioned accordingly. C and V are always cleared, and X is not affected.

MOVE TO CONDITION CODES

Mnemonic – MOVE to CCR

This is a special MOVE instruction which moves the contents of the source byte into the condition code register.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

Clearly, all the flags are affected by this operation.

MOVE TO THE STATUS REGISTER

Mnemonic – MOVE to SR

This is a special MOVE instruction which moves the contents of the source operand into the status register. This is a *Privileged Instruction* and can only be executed in supervisor mode.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	
Address register direct		

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

Clearly, this instruction will affect all bits in the status register.

MOVE FROM THE STATUS REGISTER

Mnemonic – MOVE from SR

This instruction simply stores the 16-bit status register in a specified destination operand. It is not a privileged instruction.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

No flags (or other bits in the status register) are affected.

MOVE USER STACK POINTER

Mnemonic – MOVE USP

This instruction moves the contents of the user stack pointer (A7) to or from a specified address register. It is a *Privileged Instruction* and can only be executed in supervisor mode.

MOVE ADDRESS

Mnemonic – MOVEA

This special MOVE instruction moves the contents of the source operand to a specified address register.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	
Address register direct	x	x
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

Only word and long-word operands may be specified. Word operands are sign-extended before being moved to the address register.

This instruction does not affect the flags register.

MOVE MULTIPLE REGISTERS FROM MEMORY

Mnemonic – MOVEM

This instruction causes specified registers to be loaded from consecutive memory locations beginning at the specified effective address.

The registers are loaded in the order D0 through D7, then A0 through A7. The lowest register specified is loaded from the specified effective address.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect		
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate		

In assembly language, the registers to be loaded are separated by the backslash character, thus: D1/D3/D5/A0/A2. Alternatively, a range of consecutive registers may be specified using the minus sign, thus: D0 – D5 (loads D1,D2,D3,D4,D5).

If word-size data is specified, the operands will be sign-extended before being loaded into the registers.

This instruction is used to quickly restore the status of the processor.

No flags are affected.

MOVE MULTIPLE REGISTERS TO MEMORY

Mnemonic – MOVEM

This is the complement of the previous instruction, and is used to store the processor contents quickly in memory.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect		x
Postincrement register indirect		

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

No flags are affected by this instruction.

MOVE PERIPHERAL DATA

Mnemonic – MOVEP

This instruction causes two or four bytes of data to be transferred between a specified data register and alternate-byte memory locations. This instruction is intended to simplify data transfer between the processor and 8-bit peripheral devices.

The only address mode allowed is address register indirect with displacement.

The high-order byte of the data register is transferred first, the low-order byte last. The address register is incremented by two as each byte is transferred.

No flags are affected.

MOVE QUICK

Mnemonic – MOVEQ

This instruction is a memory and speed-efficient immediate addressing move instruction for small operands (8-bit). The data can only be moved to a data register. It is sign-extended to 32 bits before being stored.

The **N** flag is set if the data is negative, the **Z** flag is set if it is zero. **C** and **V** are always cleared. **X** is not affected.

SIGNED MULTIPLY

Mnemonic – Muls

This instruction multiplies together two 16-bit operands, yielding a 32-bit result, using 2's complement signed binary arithmetic.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

The low-order word of the data register is the operand used in the multiplication. The high-order half is ignored, and is overwritten by the result.

N is set if the result is negative, cleared otherwise. Z is set if the result is zero, cleared otherwise. C and V are always cleared, X is not affected.

UNSIGNED MULTIPLY

Mnemonic – MULU

This instruction multiplies together two 16-bit operands, yielding a 32-bit result, using unsigned binary arithmetic.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

The low-order word of the data register provides the operand. The high-order word is ignored, and is overwritten by the result.

N is set if the MSB of the result is set, cleared otherwise. Z is set if the result is zero, cleared otherwise. C and V are always cleared, and X is not affected.

NEGATE DECIMAL WITH EXTEND

Mnemonic – NBCD

This instruction subtracts the destination operand and the value of the X flag from zero, and stores the result in the destination. BCD arithmetic is used.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

This instruction always operates on only one byte of data.

The C and X flags are set if a borrow occurred, cleared otherwise. The Z flag is cleared if the result is non-zero, unaltered if it is zero. The N and V flags are undefined. See ABCD for a comment on the Z flag.

NEGATE

Mnemonic – NEG

This instruction subtracts the destination operand from zero using 2's complement binary arithmetic, storing the result in the destination.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

C and X are set if a borrow occurs, cleared otherwise. N is set if the result is negative, cleared otherwise. The V flag is set if an overflow occurs, cleared otherwise.

NEGATE WITH EXTEND

Mnemonic – NEGX

This instruction subtracts the destination operand and the value of the X flag from zero, storing the result in the destination.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

This instruction is similar to NEG, but is for use in multiple-precision arithmetic.

C and X are set if a borrow occurs, cleared otherwise. The N flag is set if the result is negative, cleared otherwise. The Z flag is cleared if the result is non-zero, unchanged if it is zero. The V flag is set if there is an overflow, cleared otherwise. For a comment on the Z flag, see ABCD.

NO OPERATION

Mnemonic – NOP

Does nothing except advance the program counter. It can be used to replace instructions that are no longer needed, without having to recompute displacements, to produce a precise time delay, or to temporarily replace instructions you do not want to execute when debugging. It is rarely found in finished programs.

LOGICAL NOT (COMPLEMENT)

Mnemonic – NOT

This instruction performs a bitwise complement of the destination operand. All 0's are changed to 1's and all 1's to 0's, i.e. it is a 1's complement operation.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect	x	
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

N is set if the result is negative, cleared otherwise. Z is set if the result is zero, cleared otherwise. C and V are always cleared, and X is unaffected.

LOGICAL INCLUSIVE-OR

Mnemonic – OR

This instruction performs the inclusive-OR operation between the source data and the destination data.

Inclusive-OR is performed according to the following rules:

$$1 \text{ OR } 1 = 1$$

$$1 \text{ OR } 0 = 1$$

$$0 \text{ OR } 0 = 0$$

There are two general forms, depending whether a data register provides the source or the destination operand.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

N is set if the result is negative, cleared otherwise. Z is set if the result is zero, cleared otherwise. C and V are always cleared, and X is not affected.

INCLUSIVE-OR IMMEDIATE

Mnemonic – ORI

Performs the inclusive-OR operation between immediate data and the destination operand.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

In addition, the destination may be the condition codes or the entire status register. If it is the entire status register, this is a *Privileged Instruction*, and can only be executed in supervisor mode.

The data size may be byte, word or long-word. The immediate data supplied must match the operation size. If byte-size data is specified, the low-order (second) byte of the immediate data word is used.

The effect on the flags is the same as for OR.

PUSH EFFECTIVE ADDRESS

Mnemonic — PEA

This instruction calculates an address using one of the control addressing modes, then pushes that address on the stack.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect	x	
Postincrement register indirect		
Predecrement register indirect		
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset		
P.C. relative with index		
Immediate		

No flags are affected.

RESET EXTERNAL DEVICES

Mnemonic – RESET

This instruction simply causes a pulse on the RESET pin of the processor, as a reset signal to external devices. The only effect within the processor is to increment the program counter by 2. There is no change to other registers or memory.

ROTATE DATA REGISTER LEFT

Mnemonic – ROL

This instruction rotates the contents of a data register to the left by a number of bit positions which may be contained in another data register, or be given as immediate data. In the first case, rotations of from 0 to 63 bit positions are possible, in the latter from 1 to 8.

The bit shifted out on the left is placed in the carry flag, and it is also moved in to the rightmost bit position.

C is set according to the last bit shifted out to the left. N is set if the MSB is set, cleared otherwise. V is always cleared. Z is set if the result is zero, cleared otherwise.

ROTATE MEMORY WORD LEFT

Mnemonic – ROL

Similar to the preceding instruction, but works on a memory location. The data size is limited to word, and only rotations of one bit position are possible.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

For the effect on flags, see previous instruction.

ROTATE RIGHT (DATA REGISTER)

Mnemonic — ROR

This instruction rotates the contents of a data register to the right by a number of bit positions, counted as for ROL.

The bit rotated out on the right is placed in the carry flag, and also in the leftmost bit position.

For the effect on the flags, see ROL.

ROTATE RIGHT (MEMORY)

Mnemonic — ROR

Similar to ROR (data register) but works on one word of memory, and rotations are limited to one bit position, as for ROL memory.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

For the effect on flags, see ROL.

ROTATE LEFT WITH EXTEND

Mnemonic — ROXL

This instruction is the same as the ROL (register and memory) instructions, except that the leftmost rotated-out bits are echoed in the X flag as well as in C and the rightmost bit position. This version is intended for use in multiple-precision arithmetic.

For details of address modes, etc., see ROL (data register) and ROL (memory word).

ROTATE RIGHT WITH EXTEND

Mnemonic — ROXR

This instruction is the same as the ROR (register and memory) instructions, except that the rightmost shifted-out bits are echoed in the X flag as well as in the C flag and the leftmost bit position. This version is intended for use in multiple-precision arithmetic.

For details of address modes, etc., see ROR (data register) and ROR (memory word).

RETURN FROM EXCEPTION

Mnemonic — RTE

This instruction terminates an exception processing routine, and restores the state of the program by loading the status register, and then the program counter, from the system stack.

After the RTE instruction, the processor may be in user or supervisor mode, depending on the condition of the S bit loaded into the status register.

Obviously, all flags are affected by this instruction.

RETURN AND RESTORE CONDITION CODES

Mnemonic — RTR

This instruction is used to terminate a subroutine. It pulls a word from the stack, and places the 5 least significant bits

in the status register. Then it pulls a return address from the stack and loads it into the program counter, thus returning control to the calling program.

There is no instruction that automatically saves the condition codes on the stack when calling a subroutine. To use RTR, you must save the condition codes on the stack at the beginning of the subroutine.

Obviously, all flags may be affected by this instruction.

RETURN FROM SUBROUTINE

Mnemonic – RTS

This instruction is used to terminate a subroutine. It causes control to be returned to the calling program by pulling the return address from the stack and loading it into the program counter.

No flags are affected by RTS, thus with this return instruction the condition codes can be used to pass information from the subroutine to the calling program.

SUBTRACT DECIMAL WITH EXTEND

Mnemonic – SBCD

This instruction subtracts the source data and the value of the X flag from the destination data, using BCD arithmetic.

This instruction can use either register direct addressing, or address register indirect addressing, but both operands must use the same mode. It can therefore be used for register-from-register subtraction, or memory-from-memory subtraction. Only the least significant 8 bits of the data are affected.

The C and X flags are set if a borrow is generated, cleared otherwise. The Z flag is cleared if the result is non-zero, unchanged otherwise (see ABCD for a comment on this). N and V are undefined.

SET ACCORDING TO CONDITION

This instruction tests a specified condition code, and if the

condition is met, the contents of a (byte-size) destination operand are set to all ones. If not met, the byte is set to all zeroes.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

The following is a list of mnemonics and conditions. For an explanation of these conditions see Test Condition, Decrement and Branch.

ST	—	Set True
SF	—	Set False
SHI	—	Set High
SLS	—	Set Low or Same
SCC	—	Set Carry Clear
SCS	—	Set Carry Set
SNE	—	Set Not Equal
SEQ	—	Set Equal
SVC	—	Set oVerflow Clear
SVS	—	Set oVerflow Set
SPL	—	Set PLus
SMI	—	Set MInus
SGE	—	Set Greater or Equal
SLT	—	Set Less Than
SGT	—	Set Greater Than
SLE	—	Set Less or Equal

No flags are affected by this instruction.

LOAD STATUS REGISTER AND STOP

Mnemonic – STOP

This instruction loads the status register from 16-bit immediate data, and then stops program execution until a trace, interrupt or reset occurs. The program counter is advanced by four to point to the next instruction.

This is a *Privileged Instruction* and can only be executed while in supervisor mode.

SUBTRACT BINARY

Mnemonic – SUB

This instruction subtracts the source operand from the destination operand using binary arithmetic.

There are two general forms of this instruction, with a data register as either the source or the destination operand.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	x
Address register direct	x	x
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

Note that address register direct addressing cannot be used for byte-size operands.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	
Address register direct		
Address register indirect		x

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

C and X are set if a borrow is generated; cleared otherwise. N is set if the result is negative, cleared otherwise. V is set if an overflow occurs, cleared otherwise. Z is set if the result is zero, cleared otherwise.

SUBTRACT ADDRESS

Mnemonic – SUBA

This is a special form of SUB using an address register as the destination operand.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct	x	
Address register direct	x	x
Address register indirect	x	
Postincrement register indirect	x	
Predecrement register indirect	x	
Register indirect with offset	x	
Register indirect with index	x	
Absolute short	x	
Absolute long	x	
P.C. relative with offset	x	
P.C. relative with index	x	
Immediate	x	

If a word source data is specified instead of long-word, it will be sign-extended prior to the subtraction.

Note that unlike SUB, no flags are affected by SUBA.

SUBTRACT IMMEDIATE

Mnemonic – SUBI

This instruction subtracts immediate data from the destination operand.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		x
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

The data size can be specified as byte, word or long-word. If byte-size is specified, the low-order (second) byte of the immediate data word is used.

SUBTRACT QUICK

Mnemonic – SUBQ

This is a special version of immediate-addressing subtraction for small operands (1 to 8).

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		x
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate	x	

Note that address register direct addressing cannot be used if the operand size is byte.

The effect on flags is the same as for SUB.

SUBTRACT WITH EXTEND

Mnemonic – SUBX

This instruction subtracts both the source data and the value of the X flag from the destination data.

This instruction can use either register direct addressing or address register indirect addressing, but both operands must use the same mode. Thus it can be used for register-to-register or memory-to-memory subtraction.

When address register indirect addressing is used, both address registers are decremented before the operation. This facilitates multiple-precision arithmetic.

For the effect on the flags see SBCD.

SWAP REGISTER HALVES

Mnemonic – SWAP

This instruction exchanges the most significant 16 bits of a register with the least significant 16 bits.

The N flag will be set to the value of bit 31 after the swap. The Z flag will be set if the register contains zero. C and V are always cleared, and X is not affected.

INDIVISIBLE TEST AND SET

Mnemonic – TAS

This instruction tests a byte of data in the destination operand,

and sets the N and Z flags accordingly. It then sets the MSB of the destination data to 1. This operation cannot be interrupted.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

The C and V flags are always cleared, X is not affected.

TRAP

Mnemonic – TRAP

Initiates exception processing. The program counter is incremented to point to the next instruction, then saved on the system stack, followed by the current contents of the status register. Program execution then continues at an address obtained from the exception vector table.

A full description of this instruction and of exception processing is beyond the scope of this introductory book.

TRAP ON OVERFLOW

Mnemonic – TRAPV

This instruction will initiate exception processing if the V flag is set when it is executed.

TEST AN OPERAND

Mnemonic – TST

This instruction tests the destination operand, and sets the N and Z flags accordingly. The destination data is not altered.

<i>Address Mode</i>	<i>Source</i>	<i>Destination</i>
Data register direct		x
Address register direct		
Address register indirect		x
Postincrement register indirect		x
Predecrement register indirect		x
Register indirect with offset		x
Register indirect with index		x
Absolute short		x
Absolute long		x
P.C. relative with offset		
P.C. relative with index		
Immediate		

C and V are always cleared. X is not affected.

UNLINK

Mnemonic – UNLK

This instruction undoes the effect of a LINK instruction by loading the system stack pointer with the contents of a specified address register. This address register is then loaded with a long word pulled from the stack. See the description of the LINK instruction.

Chapter 4

ASSEMBLERS

Writing a program in assembly language is done in two stages. Firstly, the 'source code' must be written. This source code consists of the instructions we have covered in the last chapter, assembler directives, labels, and constants. Comments may also be included, but do not form part of the source code proper. This source code is written using an 'editor' (also called a screen editor or text editor), which is similar to a word processor, with facilities for entering the code, editing it, and also some sophisticated facilities like block moves and copies, and the ability to change a string of characters for another string throughout the written code.

The editor produces what is essentially a text file. The assembler proper reads this text file, and from it generates the 'object code', which is the actual machine code which will constitute the final program.

Though all assemblers will be provided with some form of editor, it is not usually necessary to use the editor supplied. You can use an editor from one assembler with another assembler, or you can use a word processor to generate the source code. In fact, if you are familiar with a particular word processor, this can be a good idea, as you do not then need to learn a whole new set of commands to use the editor.

If you use a word processor, you produce the text file for the assembler by 'printing' the document to a disc file instead of the printer. The assembler will not be able to work from a word processor work file. Also, you should not use any features such as underlining or special typefaces, or anything else which will cause printer control codes to be included in the file, as this could cause problems at the assembly stage.

There are several ways in which an assembler can work. It can take a text file in computer memory and place the object code also in memory (RAM to RAM), it can take a text file on disc and assemble the object code in memory (disc to RAM), it can take a text file in memory and assemble to disc (RAM to disc), or it can take a text file on disc and assemble

to disc (disc to disc).

Of these four options, the first is rare. It is found on a few small home coputers with built-in assemblers (e.g. BBC micro, Memotech) but is unlikely to be found on a 68000 machine. The second option is provided by some assemblers, and is useful for a quick test of small programs. The third option is very rare. The fourth option is by far the most common. It has the advantage that it allows all available memory in the computer to be used by the assembler. This means the assembler program can be more complex, which will usually mean faster assembly, and also allows longer programs, as the length is set by disc capacity not computer memory size.

Though I have referred to discs here, some machines may use alternative storage media, such as 'microdrives' on the Sinclair QL.

Assembly Language Formatting

Assembly language is written in columns. Each of these columns is described as a 'field'. Each field has a specific purpose. It is important to place things in the right fields if the assembler is to produce the object code correctly.

The first field is the 'label' field. A label gives a 'name' to a memory location, which may be used to refer to this location later in the program. You do not have to provide a label for every instruction, this column may be left blank.

The second column contains the instruction mnemonic, and is called the operation or instruction field. This is the only field which can never be blank. It must always contain either an instruction or an assembler directive.

The third column is the 'operand' or address field. An 'operand' is the data on which the instruction is to work. With some instructions and address modes, such data is inappropriate and this field will instead contain an address (which may be a label) from which the data must be fetched (or to which it must be stored) or to which the program must jump or branch, etc.

Some instructions require two operands, or one operand and an address, or two addresses. In these cases the two items

must be separated by a comma.

The last column is the 'comment' field. You may, in general, place anything in this column, as its contents are ignored by the assembler, rather like REMs in BASIC. This field is used to document the program, so that you may know the purpose of the code if you should work on it again some time after originally writing it.

The assembler needs have some way of knowing where each field begins. This varies from assembler to assembler. It may be that a fixed width is allowed for each field, and each field must be started in the correct screen column. For such assemblers, the screen editor will usually have 'tab' stops to allow easy movement to the right places.

Most assemblers, however, use 'delimiters' to separate the fields. The delimiter will be some particular character. This can vary from assembler to assembler, but by far the most common is the space. It follows that you should be careful not to put spaces where they should not be. For example, you should never put 'MOVE .W' instead of 'MOVE.W', or put 'MOVEQ 0, D0' instead of 'MOVEQ 0,D0'.

Assemblers do vary in the way they require the source code to be formatted, and you should be careful to read all the documentation with the assembler you are using. There are also differences in what will be regarded as a comment. Many assemblers allow a whole line to be a comment, usually by starting the line with an asterisk, though other symbols are used.

Labels

In essence, labels are used in assembly language to give names to memory locations. These names can be used instead of the actual addresses in branch, jump, move and many other instructions. Labels are an aid to the programmer, as a well-chosen label name is easier to remember than a string of digits, but the use of labels is also essential in producing position-independant or relocatable code, as in these cases, the actual memory locations will not be known.

Labels are used in conjunction with directives as well as instructions, and indeed labels will always be used in conjunction with EQU directives, as the purpose of these is to define a label.

The label field is the first column in an assembly language instruction. You do not have to provide a label for every line. When a label is present, the label is defined by the assembler as the address into which the first byte of the instruction will be assembled. The assembler keeps a table of these labels.

Obviously, once defined, you can use the label in another instruction's address field. However, there will be occasions when you will want to use a label in an address field before it has been defined. This occurs whenever you want to perform a forward branch.

To allow this, most assemblers use two passes of the source code. On the first pass, any references to undefined labels are assembled (usually) with the instruction's own address in the address field. On the second pass, the assembler will use the label definitions from first pass to correct these instructions. If a label is still undefined on the second pass an assembler error will be generated.

Most assemblers for the 68000 microprocessor will always do a two-pass assembly whether forward branches are present or not. However, on some you may have to specify whether you want one-pass or two-pass assembly.

There are usually rules about what you can use as labels. Often these are similar to the rules as to what you can use as a variable name in languages like BASIC. Labels must usually start with a letter, consist of letters and numerals only (occasionally other symbols may be allowed), and must normally not be the same as an instruction or directive (or in some cases begin with an instruction or directive mnemonic).

Labels often have a fixed maximum length. In other cases, the length may be unlimited (or have a long limit, like 64 characters) but only the first 5 or 6 characters may be used to distinguish between one label and another. In this latter case, it is, on the whole, better not to use more characters in a label than the computer uses.

Within these rules, it is a good idea to use 'mnemonic' labels which give some idea of what the label represents, as

this greatly improves 'readability' of a program. In particular, it is a help if you do some more work on a program some time after you originally wrote it.

Assembler Directives

In addition to the processor instruction set, most assemblers will accept 'directives', also called 'pseudo-operations'. These are similar to instructions, but instead of assembling into code to be executed by the microprocessor, they 'instruct' the assembler as to how the code is to be assembled.

These directives cover such things as the form in which the object code is to be produced (i.e. position independent, relocatable, absolute, etc.), the origin or start address if the code is to be absolute, how much work space should be allowed for the program, and similar things.

The Motorola assembler specification includes a number of these directives. Specific assemblers may include additional directives, perhaps to support machine-specific features (e.g. multi-tasking) and in some cases the scope of some directives may also be extended.

DATA. This directive is used to enable constants to be included in a program. It is used for such purposes as lookup and code conversion tables, messages and short text strings, commands and conversion factors. The simplest form is

DATA 99

which will place the value 99 in the next memory location. Most assemblers will allow more complex forms such as

DATA 99,17,23,44,43

which will place these values in the next 5 memory locations, or

DATA 'Press any key'

which will place the ASCII values of these characters in successive locations.

There may be forms of the DATA instruction for byte, word, and long word size data, or you may be able to define what size you wish to use at the start of the source code.

Some assemblers may use the form DC (for Define Constant), and this may be followed by .B, .W, or .L to define the data size. To define a string constant, you use DC.B followed by the string. If no data size is specified, word size will be assumed, but some assemblers require that a size is always specified.

There may also be a DS directive (again this will usually be followed by a size specifier). This reserves memory locations, but does not set them to initial values. You may have to specify the number of locations to be reserved in the operand field, i.e. DS.W 5 would reserve 5 contiguous word-size locations.

EQUATE or DEFINE. This directive allows names to be equated with data or addresses. EQU is the normal mnemonic for this, but DEF may also be found.

In its simplest form, the EQUATE directive assigns a numeric value to a name, as in these examples

```
START EQU 8000
TABLE EQU 9680
```

It is also possible to equate one name to another, provided the second have already been defined, or to use an equation on the right of the EQU

```
RETRY EQU START
WARMSTART EQU START+66
```

Equate directives should normally be placed at the start of the program.

ORIGIN. This directive (usual mnemonic ORG) allows the programmer to specify the memory location where a program is to start. Some assemblers also allow ORG to be used to specify the starting points of subroutines and data areas.

It is inappropriate to specify an origin if position independent of relocatable code is to be produced. Some assemblers

take the absence of an origin as an instruction to produce code in position independent form. There is a related directive **RORG** which allows a relative origin to be specified for writing code in relocatable form.

RESERVE or SIZE. This directive allows the programmer to reserve a block of memory of stated size to store the variable data which will be used by the program, i.e. data tables, temporary storage locations, a stack, buffers.

SIZE and **RESERVE** are normally not quite the same. **SIZE** just reserves a block of memory of specified size. With **RESERVE**, you can normally use a number of these to 'structure' the stored memory space, using labels.

Most assemblers will support one or the other, not both.

OTHER DIRECTIVES. Other directives are normally concerned with linking several small assembly language programs into one large one, or controlling the output of the assembler.

The linking instructions tend to vary a good deal from assembler to assembler, so no useful general information can be given here.

Most of the other directives are virtually self-explanatory. For example, **END** marks the end of the program, and **NOLIST** stops the assembler from producing a listing of the object code on the screen (or printer). When producing an output listing, there are likely to be instructions such as **PAGE**, which causes a new page to be started, and **SPACE (SPC)** which can be used to print a number of blank lines. These help to improve the appearance of the output.

Most of these directives are optional, but on many assemblers, the use of an **END** directive is obligatory.

Chapter 5

SIMPLE PROGRAMS

Addition

Though computers actually spend more time on data manipulation than on arithmetic, simple arithmetical programs are still the best introduction to assembly language programming.

This first example is to add two numbers and store the result in memory. The 68000 series processors have a whole host of addition instructions, but the most generally applicable one for binary arithmetic is **ADD**. This adds the source operand to the destination operand, storing the result in the destination operand. However, one operand must be a data register, you cannot directly add the contents of one memory location directly to another.

Adding the contents of a data register to a memory location has the advantage that you cannot accidentally forget to store the results of the addition (a common error with accumulator-based microprocessors like the 6502) but does mean that you can only use those address modes which allow memory alteration for the destination operand. Another disadvantage of this method in some circumstances is that, as the memory location used to contain one of the original numbers is used to store the result, this original number will be lost, and you may not want this to happen.

Adding the contents of a memory location to a data register allows more flexibility in the addressing modes which can be used. You do have to use **MOVE** instructions first to load one of the numbers into the register, and then to store the result, and this makes the program longer, but has the advantage that the original numbers are intact.

The 68000 series can add byte, word (16 bit) or long word (32 bit) numbers. A point to remember, however, is that if you add two byte-size numbers, the result may be too big to fit into a single byte. The same holds true for word and long word operations. However, it is not a good practice to use a larger data size for the result than you use for the sources.

In MOVE operations from memory to memory, from data registers to memory, and from memory to data registers, if you move byte-size data into a word or long word size destination, only the one byte of the destination is affected, the other byte(s) being unchanged. This can obviously lead to problems with 'left-over' values from previous operations. The best rule is always to select the data size needed for the result, and to use this size for the source data as well.

In this example, word-size data is used throughout. Since this is the default, the size specifiers could be omitted, but it is good practice always to specify the data size.

```

                ORG      $6000

NUM1    DS.W      1
NUM2    DS.W      1
RESULT  DS.W      1

START    MOVE.W    NUM1,D1      FETCH FIRST
                                   NUMBER
          ADD.W     NUM2,D1      ADD SECOND
                                   NUMBER
          MOVE.W    D1,RESULT    STORE RESULT

          RTS                                END ROUTINE

END
```

Note that an origin has been specified. This program can only be assembled as absolute code. This is because it is illegal to use program-counter relative addressing to alter the contents of a memory location, as would be done here in storing the result.

There is a way around this by using an extra instruction. This is one of the most useful and most used in the 68000 set, LEA, or 'load effective address'. This is used to load the address of the result location into one of the address registers. Address register indirect addressing can then be used to store the result. If the program is written in this way, the assembled code can be loaded and run anywhere in memory.

NUM1	DS.W	1
NUM2	DS.W	1
RESULT	DS.W	1
START	MOVE.W	NUM1,D1
	ADD.W	NUM2,D1
	LEA	RESULT,A1
	MOVE.W	D1,(A1)
	RTS	

This program is suitable for use by being 'CALL'ed from BASIC. POKEs can be used to place the numbers in the memory locations, and PEEKs to retrieve the result. The RTS ending the routine will cause a return to BASIC, but some computers will require extra instructions. For example, the Sinclair QL needs a MOVEQ #0,D0 immediately before the RTS, as if the register D0 is not zero on return to BASIC, an error is assumed, the value in D0 representing the error type.

Subtraction

You can turn the addition program into a subtraction simply by changing the ADD instruction to a SUB instruction. In this case, however, the problem of the result being larger than the original numbers does not arise.

A point should be made here about signed binary arithmetic. The 68000 processor normally assumes that all arithmetic operations are being performed on 2's complement signed binary numbers. This means that a 16-bit word will be regarded as containing numbers from -32768 to +32767 rather than from 0 to 65535. Thus there is no problem when taking a larger number from a smaller, provided you remember to treat the result as a signed binary number.

As with addition, subtraction can be performed with either a memory location or a data register as the destination, but one of the operands must be held in a data register. In this example, the destination is a memory location. Byte-size data is used, so only two memory locations are used to pass

the data. The value in NUM1 is subtracted from the value in NUM2, the result being stored in NUM2.

NUM1	DS.B	1	
NUM2	DS.B	1	
START	MOVE.B	NUM1,D1	FETCH FIRST VALUE
	LEA	NUM2,A1	LOAD ADDRESS
	SUB.B	D1,(A1)	SUBTRACT (NUM2- NUM1)
	RTS		

Note the shortness of this, despite the use of an LEA instruction to allow the code to be produced in position-independent form.

Multiplication

The 68000 instruction set includes two multiply instructions, so simple multiplication can be performed with a routine which is again very similar to the ones for addition and subtraction.

The main difference with the multiply instructions is that they only come in one data 'size'. They always take 16-bit source data, and yield a 32-bit result. You should always therefore use MOVE.W instructions to fetch the operand and MOVE.L to store the result.

The destination for a multiply instruction must be a data register. All address modes except address register direct can be used for the source data.

The two multiply instructions are MULS and MULU. The difference between them is that MULS uses signed binary arithmetic and MULU uses unsigned binary arithmetic. This means that the source data will be regarded as signed or unsigned according to the form used, and the result will be in the same form.

NUM1	DS.W	1	
NUM2	DS.W	1	
RESULT	DS.L	1	
START	MOVE.B	NUM1,D1	FETCH NUM1
	MULS	NUM2,D1	XTIPLY BY
			NUM2
	LEA	RESULT,A1	LOAD
			ADDRESS
	MOVE.L	D1,(A1)	STORE RESULT
	RTS		

This program is again written in a form which will allow it to be assembled in position-independent form.

It is possible to use this program by poking numbers into the source locations and peeking the result. It will be instructive to try this with several sets of numbers, with the multiply instruction in both MULS and MULU form. This will teach you a lot about signed and unsigned binary arithmetic.

Division

The 68000 series processors even have instructions for division. Again there are two of these, for signed (DIVS) and unsigned (DIVU) binary arithmetic. Like the multiply instructions, the data size for these instructions is fixed. A 32-bit operand in a data register is divided by a 16-bit source operand, leaving a 32-bit result in the data register. The source operand can be specified using any address mode except address register direct.

A problem with any division operation is that of division by zero. This is a mathematical paradox and simply cannot be done. If division by zero should occur with the 68000 division instructions, a TRAP occurs and the processor enters exception processing. On some machines, exception processing is reserved for the operating system and may not be used by the user. On such machines, you may need to check for a zero source operand, and abort the operation if necessary.

The simplest way of doing this is to use the TST instruction on the source operand. This will cause the Z flag to be set if the source is zero. The test can be followed by a BNE instruction (Branch if Not Equal to zero) which will cause the program to branch if the zero flag is not set. This is a good introduction to assembly language decision-making, and has been done in this example, in which NUM1 is divided by NUM2.

NUM1	DS.L	1	
NUM2	DS.W	1	
RESULT	DS.L	1	
START	MOVE.L	NUM1,D1	FETCH
			DIVIDEND
	LEA	NUM2,A1	LOAD NUM2
			ADDR
	TST	(A1)	TEST DIVISOR
	BNE	OKAY	IF NONZERO
	RTS		ABORT IF
			ZERO
OKAY	DIVS	NUM2,D1	DIVIDE
	LEA	RESULT,A2	
	MOVE.L	D1,(A2)	
	RTS		

This is slightly unsatisfactory, as if the divisor is zero, RESULT is left undefined. It would be better if some action were to be taken to indicate that the operation was unsuccessful. A simple way of doing this is to use another memory location as a flag, and set this to zero if division occurs, or to a nonzero value if the operation is aborted.

Note the use of the label 'OKAY' in this example. It is used to give a forward branch, skipping over the first RTS if the divisor is nonzero. Two-pass assembly is therefore essential in this example.

Note also how the branch instruction is used to skip over unwanted instructions when appropriate. Such an action is considered bad practice in BASIC, but in assembly language it

is perfectly acceptable, and often the best (and sometimes the only) way of arranging things.

Loops

As a first example of a loop in assembly language, we will see how to add up all the whole numbers between zero and any number up to 255. Such a program is interesting, as the same thing can be done very simply in BASIC using a FOR . . . NEXT loop. Comparing BASIC and assembly language programs doing exactly the same job is very instructive as to the difference in speeds between the two!

In essence, this program consists of loading the limiting number into a memory location, and loading the value 1 into D1. The value in D1 is added to the memory location labelled 'RESULT' (which is started at zero — note use of LEA and register indirect addressing to permit relocatable code again) and then D1 is compared to the limit. If it is equal to the limit, the loop ends. If not, the contents of D1 is increased by 1, and the program loops back to the point where D1 is added to the result. Note the use of BRA (BRanch Always) here.

An interesting point is the use of the ADDQ instruction to increment the value in D1. The 68000 does not actually have an increment instruction, found on most other microprocessors. Instead it uses this 'add quick' instruction, which can be used for small operands, from 1 to 8 only, and with immediate addressing only for the first operand.

LIMIT	DS.W	1	
RESULT	DS.W	1	
START	LEA	RESULT,A1	
	MOVE.W	#0,(A1)	INITIALISE RESULT
	MOVE.W	#1,D1	INITIALISE D1
LOOP	ADD.W	D1,(A1)	ADD D1 TO RESULT

	CMP	LIMIT,D1	CHECK FOR LIMIT
	BEQ	OUT	END IF D1= LIMIT
	ADDQ.W	#1,D1	INCREMENT D1
	BRA	LOOP	LOOP ALWAYS
OUT	RTS		ENDS ROUTINE

Clearing a Section of Memory

As a second example of a loop, we will see how to clear a section of memory, by which is meant setting all the locations to zero. You may want to do this, for example, with video memory in order to clear the screen. This example will also show further ways in which the address registers can be used.

Two address registers are used. The first is initialised with the first memory location to be cleared, the second with the last to be cleared. The 68000 instruction set includes a CLR instruction, and this is the fastest way of setting memory locations to zero.

The memory location addressed by the first address register is cleared. The register is then compared with the second register using CMPA (compare address). If the two are equal, the routine ends. If not, the first address register is incremented, and another memory location cleared.

In this example, the memory is cleared a word at a time, so the address register must be incremented by two on each pass of the loop. The long-word form of CMPA is used.

In this example, the first and last locations to be cleared are specified using the immediate addressing modes. The locations specified are, in fact, the video memory area on the Sinclair QL computer. This routine will, therefore, clear the entire screen to black on this machine. For other machines, the specified addresses can be easily altered.

```
START    MOVE.L    #$20000,A1  INITIALISE
                                           ADDRESS
```

	MOVE.L	#\$28000,A2	REGISTERS
LOOP	CLR.W	(A1)	CLEAR LOCATION
	CMPA.L	A1,A2	CHECK LIMITS
	BEQ	OUT	END IF EQUAL
	ADDQ.L	#2,A1	INCREMENT A1
	BRA	LOOP	
OUT	MOVEQ.W	#0,D0	FOR QL ONLY
	RTS		

This program demonstrates how quickly the 68000 series can alter a large chunk of memory.

Notes

Notes

Please note following is a list of other titles that are available in our range of Radio, Electronics and Computer Books.

These should be available from all good Booksellers, Radio Component Dealers and Mail Order Companies.

However, should you experience difficulty in obtaining any title in your area, then please write directly to the publisher enclosing payment to cover the cost of the book plus adequate postage.

If you would like a complete catalogue of our entire range of Radio, Electronics and Computer Books then please send a Stamped Addressed Envelope to:—

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

160	Coil Design and Construction Manual	£1.95
202	Handbook of Integrated Circuits (IC's) Equivalents & Substitutes	£1.95
205	First Book of Hi-Fi Loudspeaker Enclosures	£0.95
208	Practical Stereo and Quadrophony Handbook	£0.75
214	Audio Enthusiasts Handbook	£0.85
219	Solid State Novelty Projects	£0.85
220	Build Your Own Solid State Hi-Fi and Audio Accessories	£0.85
221	28 Tested Transistor Projects	£1.25
222	Solid State Short Wave Receivers for Beginners	£1.95
223	50 Projects Using IC CA3130	£1.25
224	50 CMOS IC Projects	£1.35
225	A Practical Introduction to Digital IC's	£1.75
226	How to Build Advanced Short Wave Receivers	£1.95
227	Beginners Guide to Building Electronic Projects	£1.95
228	Essential Theory for the Electronics Hobbyist	£1.95
RCC	Resistor Colour Code Disc	£0.20
BP1	First Book of Transistor Equivalents and Substitutes	£1.50
RP2	Handbook of Radio, TV, Ind & Transmitting Tube & Valve Equivalents	£0.60
BP6	Engineers and Machinists Reference Tables	£0.75
BP7	Radio and Electronic Colour Codes and Data Chart	£0.40
BP14	Second Book of Transistor Equivalents & Substitutes	£1.75
BP24	52 Projects Using IC741	£1.75
BP27	Chart of Radio, Electronic, Semi-conductor and Logic Symbols	£0.50
BP28	Resistor Selection Handbook	£0.60
BP29	Major Solid State Audio Hi-Fi Construction Projects	£0.85
BP32	How to Build Your Own Metal and Treasure Locators	£1.95
BP33	Electronic Calculator Users Handbook	£1.50
BP34	Practical Repair and Renovation of Colour TVs	£1.95
BP36	50 Circuits Using Germanium, Silicon and Zener Diodes	£1.50
BP37	50 Projects Using Relays, SCR's and TRIACS	£1.95
BP39	50 (FET) Field Effect Transistor Projects	£1.75
BP42	50 Simple L.E.D. Circuits	£1.50
BP43	How to Make Walkie-Talkies	£1.95
BP44	IC 555 Projects	£1.95
BP45	Projects in Opto-Electronics	£1.95
BP47	Mobile Discotheque Handbook	£1.35
BP48	Electronic Projects for Beginners	£1.95
BP49	Popular Electronic Projects	£1.95
BP50	IC LM3900 Projects	£1.35
BP51	Electronic Music and Creative Tape Recording	£1.95
BP52	Long Distance Television Reception (TV-DX) for the Enthusiast	£1.95
BP53	Practical Electronics Calculations and Formulae	£2.95
BP54	Your Electronic Calculator and Your Money	£1.35
BP55	Radio Stations Guide	£1.75
BP56	Electronic Security Devices	£1.95
BP57	How to Build Your Own Solid State Oscilloscope	£1.95
BP58	50 Circuits Using 7400 Series IC's	£1.75
BP59	Second Book of CMOS IC Projects	£1.95
BP60	Practical Construction of Pre-amps, Tone Controls, Filters & Attenuators	£1.95
BP61	Beginners Guide To Digital Techniques	£0.95
BP62	Elements of Electronics — Book 1	£2.95
BP63	Elements of Electronics — Book 2	£2.25
BP64	Elements of Electronics — Book 3	£2.25
BP65	Single IC Projects	£1.50
BP66	Beginners Guide to Microprocessors and Computing	£1.95
BP67	Counter, Driver and Numeral Display Projects	£1.75
BP68	Choosing and Using Your Hi-Fi	£1.65
BP69	Electronic Games	£1.75
BP70	Transistor Radio Fault-Finding Chart	£0.50
BP71	Electronic Household Projects	£1.75
BP72	A Microprocessor Primer	£1.75
BP73	Remote Control Projects	£1.95
BP74	Electronic Music Projects	£1.75
BP75	Electronic Test Equipment Construction	£1.75
BP76	Power Supply Projects	£1.95
BP77	Elements of Electronics — Book 4	£2.95

BP78	Practical Computer Experiments	£1.75
BP79	Radio Control for Beginners	£1.75
BP80	Popular Electronic Circuits — Book 1	£1.95
BP81	Electronic Synthesiser Projects	£1.75
BP82	Electronic Projects Using Solar Cells	£1.95
BP83	VMOS Projects	£1.95
BP84	Digital IC Projects	£1.95
BP85	International Transistor Equivalents Guide	£2.95
BP86	An Introduction to BASIC Programming Techniques	£1.95
BP87	Simple L.E.D. Circuits — Book 2	£1.35
BP88	How to Use Op-Amps	£2.25
BP89	Elements of Electronics — Book 5	£2.95
BP90	Audio Projects	£1.95
BP91	An Introduction to Radio DXing	£1.95
BP92	Easy Electronics — Crystal Set Construction	£1.75
BP93	Electronic Timer Projects	£1.95
BP94	Electronic Projects for Cars and Boats	£1.95
BP95	Model Railway Projects	£1.95
BP96	C B Projects	£1.95
BP97	IC Projects for Beginners	£1.95
BP98	Popular Electronic Circuits — Book 2	£2.25
BP99	Mini-Matrix Board Projects	£1.95
BP100	An Introduction to Video	£1.95
BP101	How to Identify Unmarked IC's	£0.65
BP102	The 6809 Companion	£1.95
BP103	Multi-Circuit Board Projects	£1.95
BP104	Electronic Science Projects	£2.25
BP105	Aerial Projects	£1.95
BP106	Modern Op-Amp Projects	£1.95
BP107	30 Solderless Breadboard Projects — Book 1	£2.25
BP108	International Diode Equivalents Guide	£2.25
BP109	The Art of Programming the 1K ZX81	£1.95
BP110	How to Get Your Electronic Projects Working	£1.95
BP111	Elements of Electronics — Book 6	£3.50
BP112	A Z-80 Workshop Manual	£2.75
BP113	30 Solderless Breadboard Projects — Book 2	£2.25
BP114	The Art of Programming the 16K ZX81	£2.50
BP115	The Pre-Computer Book	£1.95
BP116	Electronic Toys Games & Puzzles	£2.25
BP117	Practical Electronic Building Blocks — Book 1	£1.95
BP118	Practical Electronic Building Blocks — Book 2	£1.95
BP119	The Art of Programming the ZX Spectrum	£2.50
BP120	Audio Amplifier Fault-Finding Chart	£0.65
BP121	How to Design and Make your Own P.C.B.s	£1.95
BP122	Audio Amplifier Construction	£2.25
BP123	A Practical Introduction to Microprocessors	£2.25
BP124	Easy Add-on Projects for Spectrum ZX81 & Ace	£2.75
BP125	25 Simple Amateur Band Aerials	£1.95
BP126	BASIC & PASCAL in Parallel	£1.50
BP127	How to Design Electronic Projects	£2.25
BP128	20 Programs for the ZX Spectrum & 16K ZX81	£1.95
BP129	An Introduction to Programming the ORIC-1	£1.95
BP130	Micro Interfacing Circuits — Book 1	£2.25
BP131	Micro Interfacing Circuits — Book 2	£2.25
BP132	25 Simple Shortwave Broadcast Band Aerials	£1.95
BP133	An Introduction to Programming the Dragon 32	£1.95
BP134	Easy Add-on Projects for Commodore 64 & Vic-20	£2.50
BP135	The Secrets of the Commodore 64	£2.50
BP136	25 Simple Indoor and Window Aerials	£1.95
BP137	BASIC & FORTRAN in Parallel	£1.95
BP138	BASIC & FORTH in Parallel	£1.95
BP139	An Introduction to Programming the BBC Model B Micro	£2.50
BP140	Digital IC Equivalents and Pin Connections	£3.95
BP141	Linear IC Equivalents and Pin Connections	£3.95
BP142	An Introduction to Programming the Acorn Electron	£1.95
BP143	An Introduction to Programming the Atari 600XL and 800XL	£2.50
BP144	Further Practical Electronics Calculations and Formulae	£3.75
BP145	25 Simple Tropical and M.W. Band Aerials	£1.95



BERNARD BABANI BP184

An Introduction to 68000 Assembly Language

- A vast increase in running speed can be obtained when using programs written in assembly language, which in essence entails direct programming of the computer without using a high level built-in language such as BASIC. However, this can only be undertaken by someone who has a reasonable understanding of the microprocessor and some of the other hardware used in the computer, but it is not as difficult as one might think and this book tells the story.
 - The microprocessor dealt with is the 68000 series which is widely acknowledged as one of the most powerful types currently available, leading to its use in some of the latest home and business computers such as the Commodore Amiga, Atari ST range, Apple Macintosh range and the Sinclair QL etc.
-

GB £ NET +002.95

ISBN 0-85934-158-5

£2.95



9 780859 341585

Appendix to 68000 Assembly Language BP184 PVA.84.M. PENOL